Two-dimensional (2D) languages and application to handwritten graphical parsing

Bingwei Wu

Mater Research Project

Multimedia and Data Management

Academic Years 2012/2013

Two-dimensional (2D) languages and application to handwritten graphical parsing

Students:

Bingwei WU

Supervisors:

Christian VIARD-GAUDIN

Harold MOUCHERE

Two-dimensional (2D) languages and application to handwritten graphical parsing

Bingwei WU
Ecole Polytechnique de l'université de Nantes
18 août 2013

ABSTRACT

Despite the recent advances in handwriting recognition, handwritten two-dimensional (2D) languages are still a challenge. Electrical schemas, chemical equations and mathematical expressions are examples of such 2D languages. In this case, the recognition problem is particularly difficult due to the two dimensional layout of the language. The main goal of our work is to study the application of two-dimensional (2D) languages on mathematical expression recognition, which is a special case of 2D graphical documents. The research work will be focus on context-free grammars which has the potential to cope with structural relations in documents.

The first part of this report gives an overview of mathematical expression recognition as well as different kinds of grammars applied in the recognition. The second part of the report presents our developed system, including grammars, segmentation hypothesis generator, parsing algorithm and spatial relation.

Keywords: Pattern recognition, Graphical languages, Two-dimensional grammars, Handwriting recognition.

Table of Contents

Chapter 1. Introduction

1.1 Background

Currently, digital devices such as smart phone are popular all over the world. As the digital devices are increasingly used, handwritten online documents are emerging. The rapid increase in the number of online handwritten documents leads to mounting pressure on finding new solutions for faster processing, retrieval and recognition. Researchers found that 2D graphical languages [1] have the potential to cope with the structural relation in 2D documents. As a result, the application of 2D languages on handwritten graphical parsing becomes a popular field of study.

Handwriting recognition is considered as a complex field of pattern recognition and it has been addressed with a series of more and more complex challenges. First, limited to the recognition of isolated symbols, it has been then extended to deal with non-constrained handwriting where at the same time segmentation and recognition issues have to be considered. With the introduction of statistical model languages [2], it has been possible to go beyond word recognition with efficient solution for text recognition. However, in all of these cases, a strong assumption is used. The input can be considered as a global one dimensional (1D) layout of symbols forming words and then texts. This is no more the case, if we want to process structural information such as tables, diagrams, mathematical expressions, etc. where the layout conveys as much information as the symbols themselves. Hence, despite the recent advances in handwriting recognition, handwritten two-dimensional (2D) languages are still a challenge. The problems appear very complex and cannot be resolved with tools dedicated to 1D languages such as textual languages.

1.2 Scope and Outline

Our work is to study mathematical expression recognition, which is a special case of 2D graphical documents. The emphasis is on two dimensional languages. There are many kinds of 2D languages, but we are more interested in context-free grammar because it has the potential to cope with structural relations in documents. Although our study is on mathematical expression recognition, our work is not only restricted on mathematical expression. It can extend to other kinds of graphical documents.

This report is divided into five chapters. The first chapter presents background and scope of our work. In chapter 2, we give an overview of mathematical expression

recognition techniques as well as two-dimensional (2D) languages and its application on mathematical expression recognition. In chapter 3, we described our developed system. In chapter 4, the results of experiments on two databases are reported. Finally, we make a conclusion of our work and point out the future work.

Chapter 2. State of the Art

2.1 Overview of Mathematical Expression Recognition

Mathematical expression plays an important role in scientific issues as well as many other documents. However, the input of mathematical expressions is not easy because they consist of many special symbols like Greek letters and operators. Currently, many useful tools (e.g., LaTex) support the input of mathematical expressions into digital documents. However, working with this kind of tool requires special skill and training. The most natural way for human beings to produce mathematical expressions is writing. Consequently, the recognition of mathematical expressions is worthy of further study.

Mathematical expression recognition can be categorized from different point of views.

● Printed versus Handwritten

Printed expressions are formal and more regular. Handwritten expressions are more difficult to be recognized because different people have different writing styles.

● On-line versus Off-line

On-line recognition considers the time information of pen strokes. The mathematical expression is given as a sequence of sample points. Off-line recognition does not consider any time information. The mathematical expression is given as only an image.

The problem of mathematical recognition is usually divided into three stages [6]: segmentation, symbol recognition and interpretation (structural and syntactic analysis). In the stage of segmentation and symbol recognition, the expression is segmented and each segment is recognized as a symbol. In the stage of interpretation, the structure of expression is analyzed. For example, given a simple expression "$x^2 = 1$", the place of the symbol "2" need to be consider: it should be placed as the upper right (superscript) of x, or the right of x.

Mathematical expression recognition consists of three major stages: segmentation, symbol recognition and interpretation (structural and syntactic analysis). Figure 2.1 shows the architecture of recognition. The first step is to segment the mathematical expression into groups. Each of these groups forms a single symbol. In the second step, a classifier is needed to recognize each of the segments. After the recognition step, a list of objects with attributes (e.g., location, size, and probability, etc.) are returned. Finally, we apply structural analysis to obtain the structure of the expression.

Typically, the above three stages are implemented step by step. In this way, an error occurring in one step would be inherited by the following steps. Furthermore, the whole context resolves local ambiguities, and enables robust recognition. As a result, some systems adopted a global approach [6, 7, 8]: they implement these three stages simultaneously.

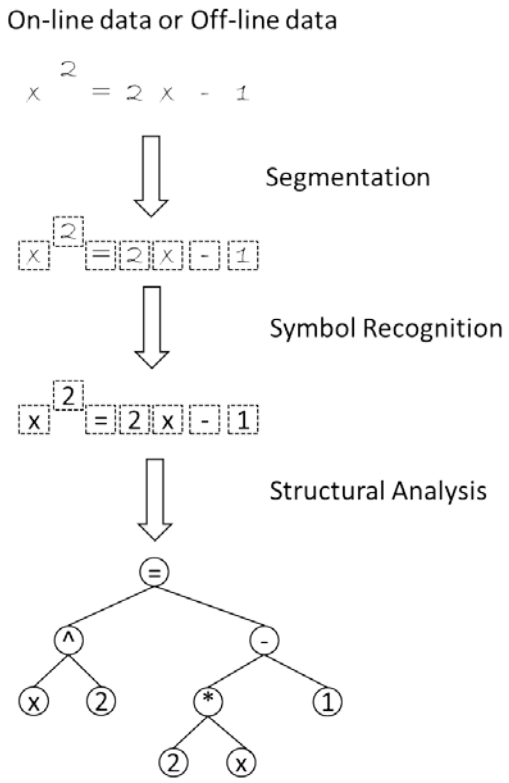


Figure 2.1 Architecture of mathematical expression recognition

As stated above, mathematical expression recognition has different categories and many problems need to be considered. In [6], an overview of mathematical expression recognition problem is given. [7] is a survey of existing works. It provided a comparison between different systems.

In Section 2.3, we will review some existing work on interpretation stage. In particular, we will highlight the similarities and differences between different approaches. In addition, a comparison of other two stages can be found in [5].

2.2 Language and Grammar

In mathematics, computer science and linguistics, a language (when the context is not given, often called a formal language for clarity) is a set of strings of symbols that may be constrained by rules that are specific to it. The alphabet of a formal language is the set of symbols, letters, or tokens from which the strings of the language may be formed; frequently it is required to be finite. A

formal language is often defined by means of a formal grammar such as a regular grammar or context-free grammar.

A formal grammar is a set of production rules for strings in a formal language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar does not describe the meaning of the strings or what can be done with them in whatever context.

A formal grammar is a set of rules for rewriting strings, along with a "start symbol" from which rewriting starts. Therefore, a grammar is usually thought of as a language generator. However, it can also sometimes be used as the basis for a "recognizer": a function in computing that determines whether a given string belongs to the language or is grammatically incorrect.

Parsing is the process of recognizing an utterance (a string in natural languages) by breaking it down to a set of symbols and analyzing each one against the grammar of the language. Most languages have the meanings of their utterances structured according to their syntax—a practice known as compositional semantics. As a result, the first step to describing the meaning of an utterance in language is to break it down part by part and look at its analyzed form (known as its parse tree in computer science, and as its deep structure in generative grammar).

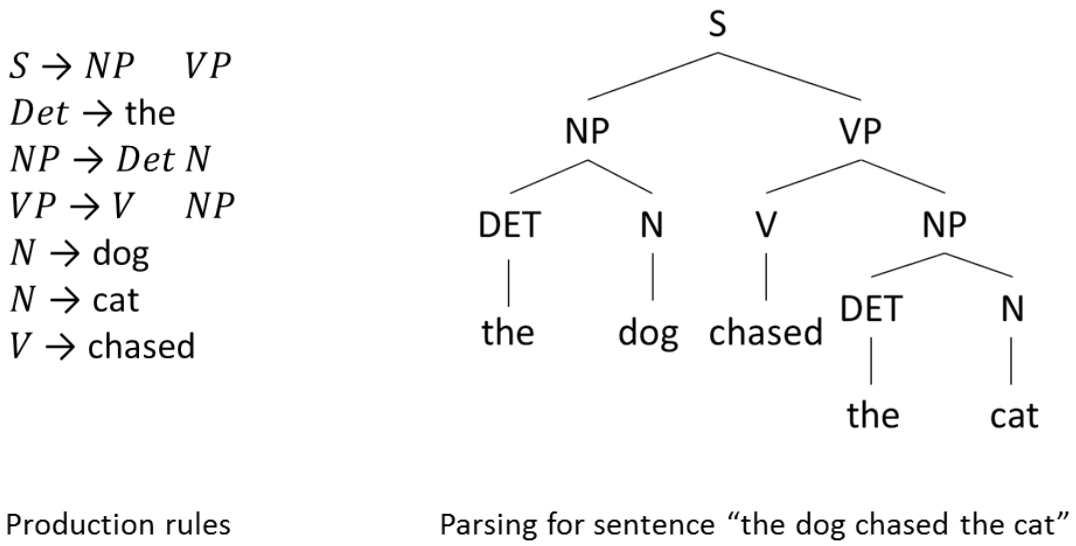


Figure 2.2 Example of parsing a sentence “the dog chased the cat”

Figure 2.2 shows a simple example of parsing a sentence. Given a set of production rules on the left hand side, we firstly extract the tokens from the sentence “the dog chased the cat”. Then we go through the tokens. “the” is a determiner (denoted by DET), “dog” is a noun (denoted by N), “chased” is a verb (denoted by V), the second “the” is a DET, “cat” is a N. Next, a DET and a N can form to be a NP (Noun Phrase), a V and a NP can form to be a VP (Verb Phrase). Finally, they form to be a start symbol S.

In the classic formalization of generative grammars first proposed by Noam Chomsky in the 1950s [3], a grammar G consists of the following components:

● A finite set N of nonterminal symbols, none of which appear in strings formed from grammar G.
● A finite set $\sum$ of terminal symbols that is disjoint from N.
● A finite set P of production rules, each rule of the form

$$(\sum \cup N)^* N (\sum \cup N)^* \longrightarrow (\sum \cup N)^*$$

where $*$ is the Kleene star operator and $\cup$ denotes set union. That is, each production rule maps from one string of symbols to another, where the first string contains an arbitrary number of symbols provided at least one of them is a nonterminal. In the case that the second string consists solely of the empty string (that is contains no symbols at all), it may be denoted with a special notation (often ε) in order to avoid confusion.
● A distinguished symbol $S \in N$ that is the start symbol.

A grammar is formally defined as the tuple $(\mathrm{N}, \sum, \mathrm{P}, \mathrm{S})$. Nonterminals are often represented by uppercase letters, terminals by lowercase letters, and the start symbol by S. For example, the grammar with terminals $\{a, b\}$, nonterminals $\{S, A, B\}$, production rules:

$$S \longrightarrow ABS$$
$$S \longrightarrow \varepsilon$$
$$BA \longrightarrow AB$$
$$BS \longrightarrow b$$
$$Bb \longrightarrow bb$$
$$Ab \longrightarrow ab$$
$$Aa \longrightarrow aa$$

and start symbol S, defines the language of all words of the form $a^n b^n$ (i.e. n copies of a followed by n copies of b).

When Noam Chomsky first formalized generative grammars in 1956 [3], he classified them into types now known as the Chomsky hierarchy. The Chomsky hierarchy consists of the following levels (see Table 2.1).

Table 2.1 Summary for Chomsky hierarchy

Grammar	Languages	Production rules
Type-0	Recursively enumerable	$\alpha \rightarrow \beta$ (no restrictions)
Type-1	Context-sensitive	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	$A \longrightarrow \gamma$
Type-3	Regular	$A \longrightarrow a$ and $A \longrightarrow aB$

Type-0 grammars (unrestricted grammars) include all formal grammars. The grammar rules have no restriction. They generate exactly all languages that can be recognized by a Turing machine. These languages are also known as the recursively enumerable languages.

Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages. These grammars have rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$ with A a nonterminal and α, β and γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be nonempty. The rule $S \longrightarrow \varepsilon$ is allowed if S does not appear on the right side of any rule.

Type-2 grammars (context-free grammars) generate the context-free languages. These are defined by rules of the form $A \longrightarrow \gamma$ with A a nonterminal and γ a string of terminals and nonterminals. Context-free languages – or rather the subset of deterministic context-free language – are the theoretical basis for the phrase structure of most programming languages, though their syntax also includes context-sensitive name resolution due to declarations and scope. Often a subset of grammars are used to make parsing easier, such as by an LL parser.

Type-3 grammars (regular grammars) generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed by a single nonterminal (right regular). Alternatively, the right-hand side of the grammar can consist of a single terminal, possibly preceded by a single nonterminal (left regular); these generate the same languages – however, if left-regular rules and right-regular rules are combined, the language need no longer be regular. The rule $S \longrightarrow \varepsilon$ is also allowed here if S does not appear on the right side of any rule.

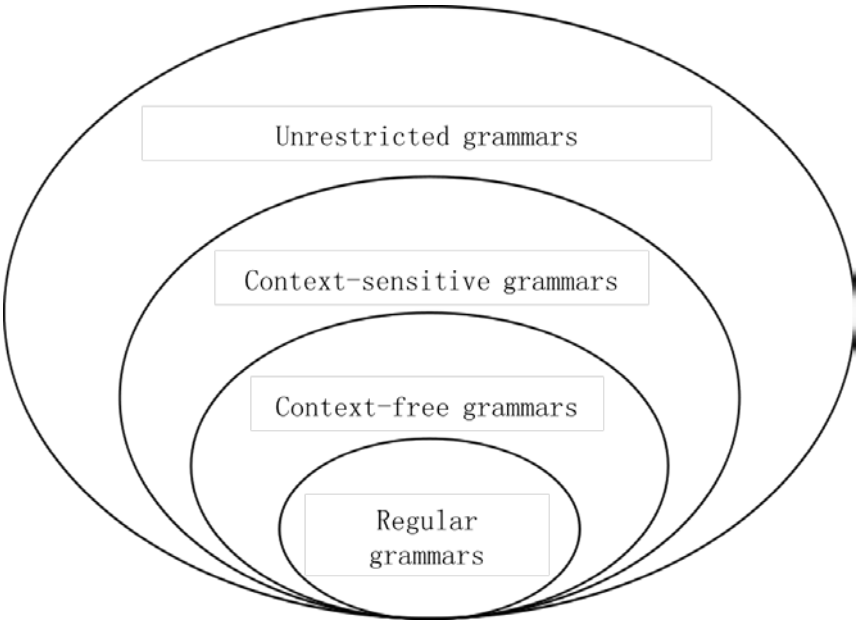


Figure 2.3 Comparison between different types of grammars

Comparison between different types of grammars in Chomsky hierarchy is shown as Figure 2.3. The difference between these types is that they have increasingly strict production rules and can express fewer formal languages. Two important types are context-free grammars (Type 2) and regular grammars (Type 3). The languages that can be described with such a grammar are called context-free languages and regular languages, respectively. Although much less powerful

than unrestricted grammars (Type 0), these two restricted types of grammars are most often used because parsers for them can be efficiently implemented.

2.3 Grammars in Recognition

Two-dimensional grammar is a common strategy for analyzing mathematical expression structure. Grammars rules are used to define the grouping of individual symbols, and to define the spatial meaning of grouping.

2.3.1 Two-dimensional Context-free Grammar

Several approaches work on two-dimensional context-free grammars [9, 10].

Průša and Hlaváč [9] use two-dimensional context-free grammars (2D-CFG) to model off-line handwritten mathematical formulae. In their work, each production rule is transformed into Chomsky Normal Form and associated with a spatial relation (denoted by spr).

As is described in Section 2.2, a context-free grammar (CFG) is a tuple (V_N, V_T, S_0, P), where:

(1) V_N is a finite set of nonterminals;

(2) V_T is a finite set of terminals;

(3) S_0 is the initial nonterminal;

(4) P is a finite set of productions: $A \to \alpha$, where $A \in V_N$ and $\alpha \in (V_N \cup V_T)^+$.

A CFG in Chomsky Normal Form (CNF) is a CFG in which the production rules are of the form $A \to BC$ or $A \to a$ where $A, B, C \in V_N$ and $a \in V_T$. Every grammar in CNF is context-free, and every CFG can be transformed into an equivalent one which is in CNF.

In the grammars defined by Průša and Hlaváč, each production rule is transformed into CNF and associated with a spatial relation (denoted by spr). The grammar is in the formalization as below.

$$A \to t$$

$$A \xrightarrow{spr} BC$$

$$A, B, C \in V_N,\ t \in V_T$$

$$spr \in \{up,\ bottom,\ left,\ right,\ subscript,\ superscript,\ inside\}$$

The term spr describes the spatial relation between nonterminals. The term spr, which only exists in 2D-CFG, is the difference between 2D case and 1D case. The terminal productions do not contain spatial relationship spr because there is no spatial relationship with only one symbol.

The parsing of 2D-CFG is similar with that of 1D case shown in Figure 2.2. In order to illustrate 2D-SCFG, we present a simple example. Given a simple grammar (V_N, V_T, S_0, P) where:

$$V_N = \{Exp, OpExp, Alphabet, Num, Op\}$$
$$V_T = \{a, b, c, 0,1,2,3,4,5,6,7,8,9, +, -\}$$
$$S_0 = Exp$$
$$spr = \{horizontal, superscript\}$$

the production rules P are:

$$Exp \xrightarrow{horizontal} Exp\ OpExp$$

$$Exp \xrightarrow{superscript} Alphabet\ Num$$

$$OpExp \xrightarrow{horizontal} Op\ Exp$$

$$Exp \rightarrow Alphabet$$
$$Exp \rightarrow Num$$
$$Num \rightarrow [0,1,2,3,4,5,6,7,8,9]$$
$$Alphabet \rightarrow [a, b, c]$$

Given an expression "$a^3 + b^2 + 1$", we firstly extract the tokens from the expression. Then we go through the tokens. "a" is a $Alphabet$, "3" is a Num, "+" is an operator (denoted by Op), "2" is a Num, "+" is a Op, "1" is a Num. Next, we combine the nonterminals according to the production rules, until we reach a start symbol Exp. The parsing tree is shown as Figure 2.4.

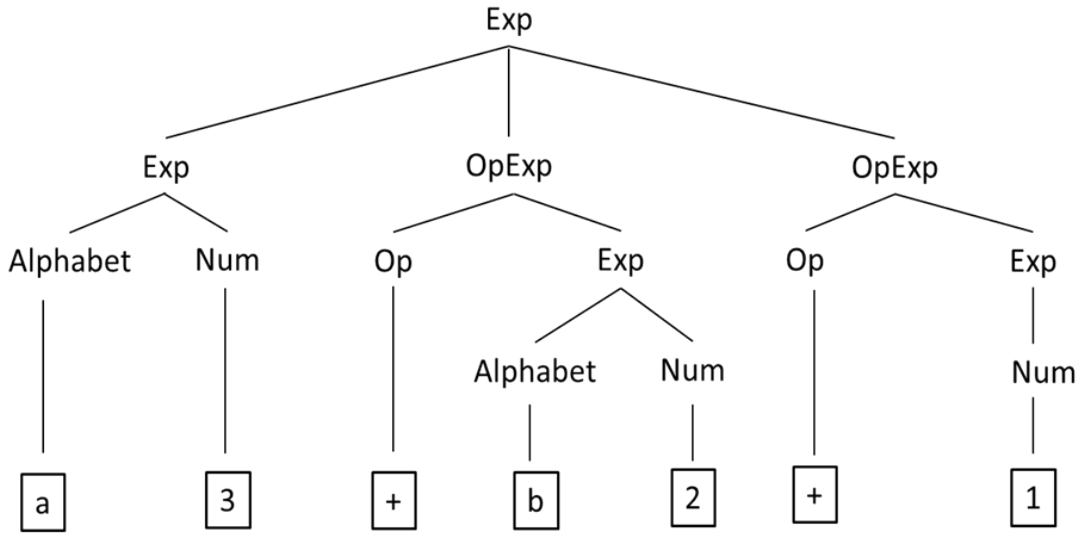


Figure 2.4 Parsing tree for $a^3 + b^2 + 1$

The structural analysis in [9] is penalty oriented. The formula structure with least penalty is the desired structure. It can be effectively parsed thanks to constraints defined via rectangles and the usage of orthogonal range searching. Time complexity is lower than CYK algorithm because it does not process all rectangles in the input. The other novelty is not treating symbol segmentation and structural analysis as two separate processes. This allows the system to recover from errors made in initial symbol segmentation.

To deal with on-line data, Průša and Hlaváč [10] propose an elementary symbols detection different from [9]. The data structure and parsing algorithm in structural analysis are modified.

2.3.2 Two-dimensional Stochastic Context-free Grammar

Yamamoto et al. [7] and Álvaro et al. [12] used 2D stochastic context-free grammar to model the spatial relations between symbols in mathematical expressions.

A stochastic context-free grammar (SCFG; also probabilistic context-free grammar, PCFG) is a context-free grammar in which each production is augmented with a probability. So every rule is associated a probability:

$$P(r_i) = P(A \to \alpha_i) \in [0,1]$$

For $\forall A \in V_N$, $\sum_{i=1}^{n_A} P(A \to \alpha_i) = 1$ where n_A is the number of rules associated to non-terminal symbol A.

The rules in 2D-SCFG are defined as $\mathrm{A} \xrightarrow{spr} \alpha$, where $\mathrm{A} \in V_N$, $\alpha \in (V_N \cup V_T)^*$ and *spr* denotes the spatial relationship that the rule models. The possible spatial relationships are: up, bottom, left, right, superscript, subscript and inside. Similar with 2D-CFG, the grammar rules can be represented in CNF as follow:

$$A \to t, Pr(A \to t)$$

$$A \xrightarrow{spr} BC, Pr\left(A \xrightarrow{spr} BC\right)$$

$$A, B, C \in V_N, t \in V_T$$

$$spr \in \{up,\ bottom,\ left,\ right,\ subscript,\ superscript,\ inside\}$$

where $Pr(A \to t)$ and $Pr\left(A \xrightarrow{spr} BC\right)$ are the probability of production rules.

The probability of a derivation (parse) is then the product of the probabilities of the productions used in that derivation. With the probability, Yamamoto et al. [7] formulated the recognition problem as a search problem of the most likely mathematical expression candidate, which can be solved using the CYK algorithm. Figure 2.5 shows a simple example of Yamamoto's algorithm. The main disadvantage of this algorithm is its dependency with respect to the temporal order of strokes. As a result, the user must input strokes in a correct order pre-processing methods must be applied.

Compared with 2D-CFG, 2D-SCFG is a probabilistic model because of associating each production rule with a probability. It makes it possible to apply machine learning to automatically learn the grammar from a training dataset [8].

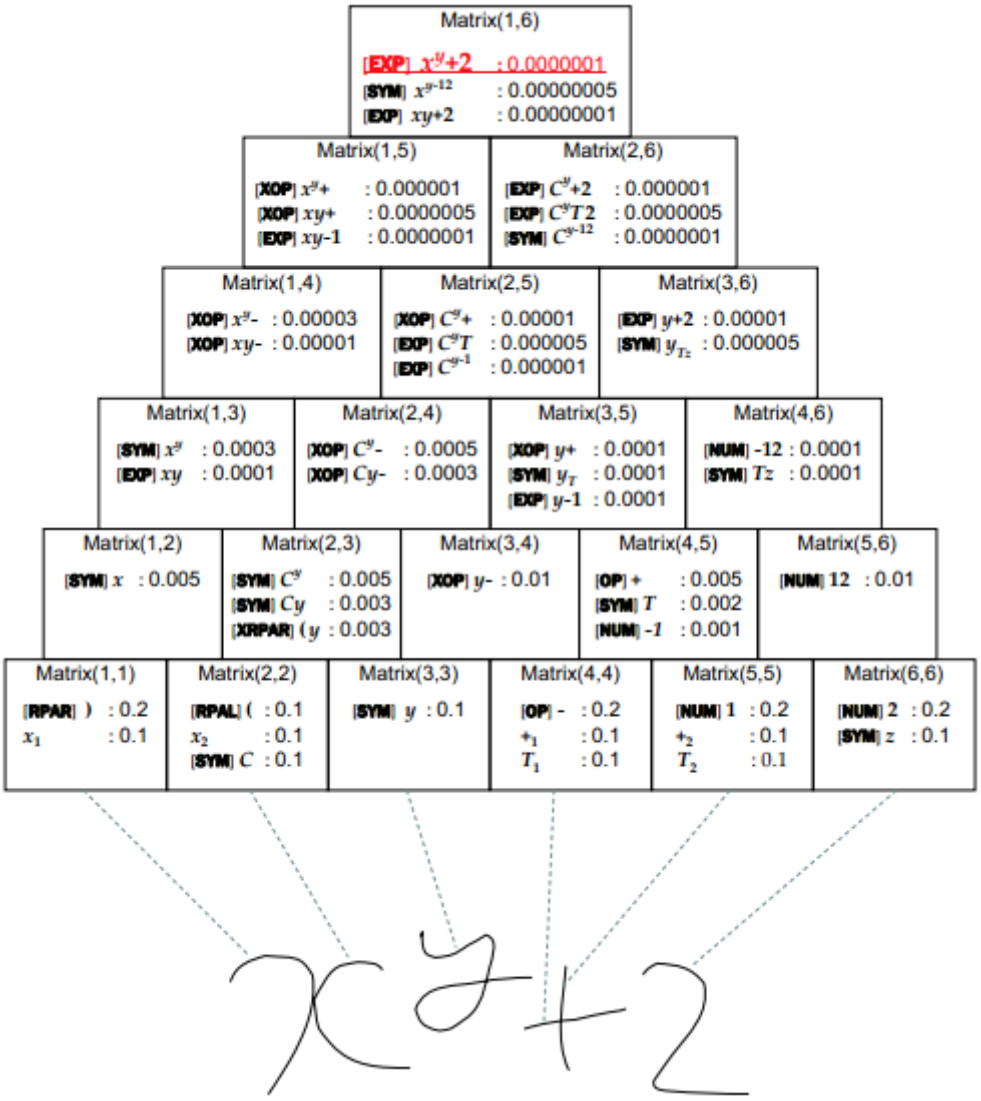


Figure 2.5 Example of a search for most likely expression candidate using the CYK algorithm

2.3.3 Fuzzy Relational Context-Free Grammar

Scott et al. [13] and MacLean[21] used a variant of the relational context-free grammar called fuzzy relational context-free grammar (Fuzzy r-CFG) to model mathematical structure. Fuzzy r-CFG is similar to SCFG. Instead of the probability $Pr\left(A \xrightarrow{spr} BC\right)$, each production rule is associated to a fuzzy function r_T. And fuzzy function is to represent the degree to which a particular interpretation of expression is valid.

The advantage of Fuzzy r-CFG is that it provides confidence score for each interpretation. But the disadvantage is that it is not a probabilistic model as SCFG, and it cannot apply statistical method.

2.3.4 Other Grammars

In addition to context-free grammars, graph grammar, or graph rewriting, is also a general technique in structural analysis. Graph grammar has been applied to mathematical expression recognition as [19]. The initial graph contains nodes to represent each symbol and contains no edges. Each single node is associated with attribute recording location and meaning of each symbol. Then graph rules are applied to add edges representing spatial relations between symbols. The output of a successfully recognized expression is a single node whose attribute represents the high-level meaning of the input expression as a character string. Graph grammars encode the spatial relation into topology of graph. However, its calculation time is long.

Another grammar is geometric grammar [20]. It describes how a geometry or structure can be generated. However, it only concerns the structure information, not taken into account syntax.

M. Shilman and P. Viola [14] present a grouping method in handwritten text in diagrams and equations, based on neighborhood graph. Then dynamic programming is used to search for the minimum cost interpretation. A. Kosmala and G. Rigoll [15] use context dependent graph grammars, and reduce the parsing complexity from O(n^2) to O(n) by optimizing graph parsing.

2.4 Summary

As mentioned above, many grammars are applied in mathematical expression interpretation. Table 2.2 shows a summary.

2D grammars are a powerful tool. However, they only describe the spatial relations roughly (right, top, bottom, superscript, subscript, and inside). To describe the relative position more precisely, some works combine grammar productions with statistical model. The model is learned from a training database.

Stria et al. [11] combined 2D grammar with statistical model of individual symbol relationship. In the schema, distributions are obtained by learning spatial relation from the dataset. And then relative values between two nodes in the relational tree are computed. Appropriate relationships are selected when their distributions fit the values.

Álvaro et al. [8] train a Support Vector Machine (SVM) classifier using a set of features describing the spatial relations. They learn the spatial relations distribution from training data. From the resulting trained SVM models, the probabilities in the parsing are able to be computed.

The approach proposed by Awal et al. [6] allows learning spatial relations directly from complete expressions. Each production rule of the grammar is associated to a Gaussian model specific to each spatial relation.

In addition to 2D grammars, other proposals are presented. Miller and Viola [17] proposed a geometric approach which uses convex hulls for grouping symbols and geometrical data structure to control the complexity of parsing. Liang et al. [18] improve Miller and Viola's algorithm and introduce several new types of geometrical data structures (e.g., Rectangle Hull Region, Convex Hull Region, Graph Region and Partial Order Region etc.) to speed up the parsing.

Table 2.2 Summary of different grammars in Mathematical expression recognition

Method	Reference paper	Author
2D Context-Free Grammar (2D-CFG)	2D Context-Free Grammars: Mathematical Formulae Recognition [9], 2006	D. Průša and V. Hlaváč
	Structural Construction for On-Line Mathematical Formulae Recognition [10], 2008	D. Průša and V. Hlaváč
Fuzzy Relational Context-Free Grammar (Fuzzy r-CFG)	Structural analysis of handwritten mathematical expressions through fuzzy parsing [22], 2006	J. Fitzgerald
	Recognizing handwritten mathematics via fuzzy parsing [21], 2010	S. MacLean and G. Labahn
2D Stochastic Context-Free Grammar (2D-SCFG)	On-Line Recognition of Handwritten Mathematical Expressions Based on Stroke-Based Stochastic Context-Free Grammar [7], 2006	R. Yamamoto and S. Sako
	A global learning approach for an online handwritten mathematical expression recognition system [6], 2012	A. Awal, H. Mouchère, and C. Viard-Gaudin
	Recognition of On-line Handwritten Mathematical Expressions Using 2D Stochastic Context-Free Grammars and Hidden Markov Models [8], 2012	Francisco Álvaro, Joan-Andreu Sánchez
Graph Grammar	On-Line Handwritten Formula Recognition using Hidden Markov Models and Context Dependent Graph Grammars [15], 1999	A. Kosmala and G. Rigoll
	Mathematics recognition using graph rewriting [19], 1995	A. Grbavec and D. Blostein
Geometric Grammar	Using a generic document recognition method for mathematical formulae recognition [20], 2002	P. Garcia and B. Coüasnon

Chapter 3. Developed System

This chapter describes the developed system for parsing mathematical expressions. The architecture is shown as Figure 3.1. The input is InkML file. Segmentation hypothesis generator is used to generate segments. MLP is used as symbol classifier. The Interpretation is carried out based on CYK algorithm with the help of 2D stochastic context-free grammar and spatial relation. The output is in both LaTex and MathML format.

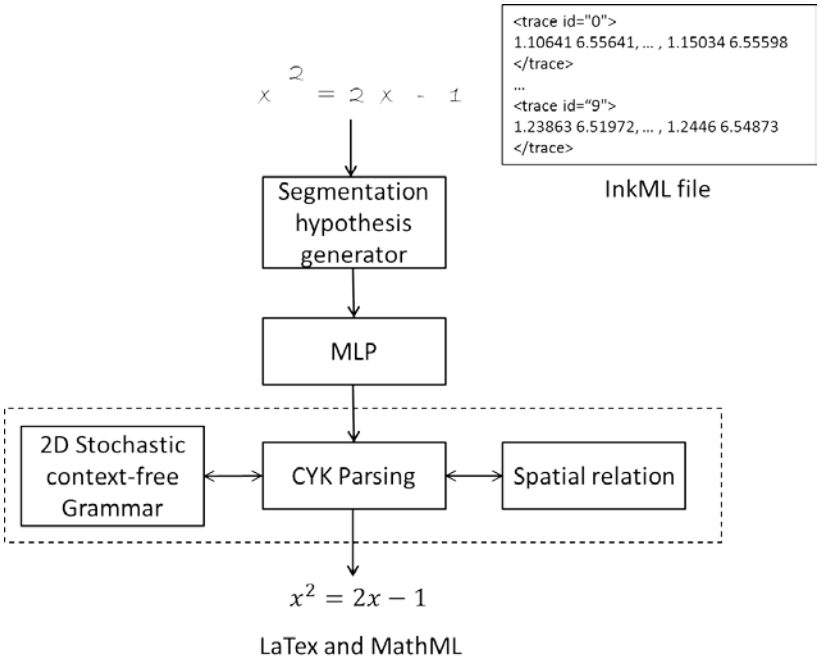


Figure 3.1 Architecture of our developed system

3.1 Mathematical Expression Grammar

A two-dimensional stochastic context-free grammar is used in our system. This grammar is defined manually trying to cover a wide range of expressions. We tried to model all the expressions that appear in the dataset in our experiment. There are six kinds of spatial relation defined in our grammar: subscript (Sub), superscript (Sup), horizontal (H), vertical (V), sub-super-expression (SSE) and inside (Ins). However, the grammar cannot parse the cases like left subscript, left superscript (${}_1^2a$) or matrix. But these kinds of expression do not appear in our experiment dataset.

Table 3.1 shows all the non-terminals in our grammar. There are totally 21 non-terminals where initial state must be *Exp* or *Sym*.

Table 3.1 Non-terminals in our grammar

Nonterminals	**Initial State**
Exp, Sym, ExpOp, OpUn, ROpUn, OBExp, OpBin, OverExp, Over, OverSym, LeftPar, RightPar, RPExp, SSExp, BigOpExp, BigOp, Sqrt, Func, 2Let, Let, SupSym	Exp, Sym

Table 3.2 shows all the nonterminal production rules that we defined. Each production rule consists of six parts: probability, spatial relation, one father (left-hand side, denoted by A), two children (right-hand side, denoted by B and C), LaTex output format. For example, the production rule

1.0 Sub Exp -> BigOp Exp "$1_{$2}"

represents that an *Exp* can be obtained by the combination of a *BigOp* and an *Exp* with the spatial relation of *Sub* (subscript). The probability of this production rule is 1.0. The Latex output format is *"$1_{$2}"* where *$1*, *$2* represent the first child and second child, respectively.

In the production rules of vertical relation (*V*), there is an additional merge flag representing how father define its reference line. For example, in the production rule

1.0 V Exp -> Exp OverExp "\frac{$1}{$2}" BCC

Merge flag "BCC" represents that father (*Exp*) uses sup-line of A (*Exp*), and center-line and sub-line of B (*OverExp*). This will be described in detail in section 3.5.

Table 3.2 Stochastic context-free grammar used in our system

Prob	Relation	A	->	B	C	Latex Format	Merge Flag
1.0	Sup	Exp	->	ExpOp	SupSym	"{$1}^{$2}"	
1.0	Sup	Exp	->	Sym	SupSym	"{$1}^{$2}"	
1.0	Sub	Exp	->	BigOp	Exp	"$1_{$2}"	
1.0	Sub	Exp	->	BigOp	Sym	"$1_{$2}"	
1.0	H	Exp	->	Exp	Exp	"$1 $2"	
1.0	H	Exp	->	Exp	Sym	"$1 $2"	
1.0	H	Exp	->	Sym	Exp	"$1 $2"	
1.0	H	Exp	->	Sym	Sym	"$1 $2"	
1.0	H	Exp	->	OpUn	Exp	"$1 $2"	
1.0	H	Exp	->	OpUn	Sym	"$1 $2"	
1.0	H	Exp	->	Exp	ROpUn	"$1 $2"	
1.0	H	Exp	->	Sym	ROpUn	"$1 $2"	
1.0	H	Exp	->	Exp	OBExp	"$1 $2"	
1.0	H	Exp	->	Sym	OBExp	"$1 $2"	
1.0	H	OBExp	->	OpBin	Exp	"$1 $2"	
1.0	H	OBExp	->	OpBin	Sym	"$1 $2"	
1.0	Sup	Exp	->	ExpOp	Exp	"{$1}^{$2}"	
1.0	Sup	Exp	->	ExpOp	Sym	"{$1}^{$2}"	
1.0	Sup	Exp	->	Sym	Exp	"{$1}^{$2}"	

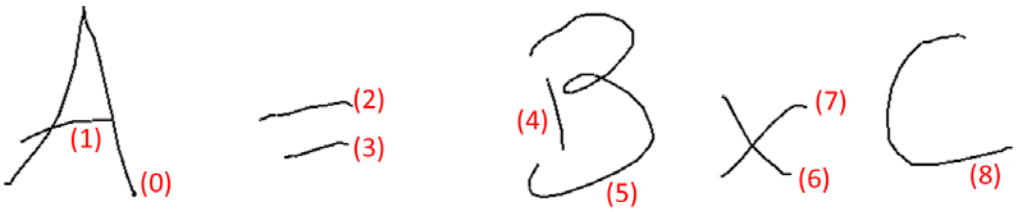
1.0	Sup	Exp	->	Sym	Sym	"{$1}^{$2}"	
1.0	Sub	Exp	->	ExpOp	Exp	"{$1}_{$2}"	
1.0	Sub	Exp	->	Sym	Exp	"{$1}_{$2}"	
1.0	Sub	Exp	->	ExpOp	Sym	"{$1}_{$2}"	
1.0	Sub	Exp	->	Sym	Sym	"{$1}_{$2}"	
1.0	V	Exp	->	Exp	OverExp	"\frac{$1}{$2}"	BCC
1.0	V	Exp	->	Sym	OverExp	"\frac{$1}{$2}"	BCC
1.0	V	OverExp	->	Over	Exp	"$2"	BBC
1.0	V	OverExp	->	Over	Sym	"$2"	BBC
1.0	H	Exp	->	LeftPar	RPExp	"$1 $2"	
1.0	H	ExpOp	->	LeftPar	RPExp	"$1 $2"	
1.0	H	RPExp	->	Exp	RightPar	"$1 $2"	
1.0	H	RPExp	->	Sym	RightPar	"$1 $2"	
1.0	H	Exp	->	ExpOp	SSExp	"{$1}$2"	
1.0	H	Exp	->	Sym	SSExp	"{$1}$2"	
1.0	H	Exp	->	BigOp	SSExp	"$1$2"	
1.0	SSE	SSExp	->	Exp	Exp	"_{$2}^{$1}"	
1.0	SSE	SSExp	->	Sym	Exp	"_{$2}^{$1}"	
1.0	SSE	SSExp	->	Exp	Sym	"_{$2}^{$1}"	
1.0	SSE	SSExp	->	Sym	Sym	"_{$2}^{$1}"	
1.0	SSE	SSExp	->	SupSym	Exp	"_{$2}^{$1}"	
1.0	SSE	SSExp	->	SupSym	Sym	"_{$2}^{$1}"	
1.0	SSE	SSExp	->	Exp	SupSym	"_{$2}^{$1}"	
1.0	SSE	SSExp	->	Sym	SupSym	"_{$2}^{$1}"	
1.0	SSE	SSExp	->	SupSym	SupSym	"_{$2}^{$1}"	
1.0	V	Exp	->	Exp	BigOpExp	"$2^{$1}"	CCC
1.0	V	Exp	->	Sym	BigOpExp	"$2^{$1}"	CCC
1.0	V	BigOpExp	->	BigOp	Exp	"$1_{$2}"	BBB
1.0	V	BigOpExp	->	BigOp	Sym	"$1_{$2}"	BBB
1.0	H	Exp	->	BigOpExp	Exp	"$1 $2"	
1.0	H	Exp	->	BigOpExp	Sym	"$1 $2"	
1.0	Ins	Exp	->	Sqrt	Exp	"\sqrt{$2}"	
1.0	Ins	Exp	->	Sqrt	Sym	"\sqrt{$2}"	
1.0	H	Exp	->	Exp	Func	"$1 $2"	
1.0	H	Exp	->	Sym	Func	"$1 $2"	
1.0	H	Exp	->	Func	Exp	"$1 $2"	
1.0	H	Exp	->	Func	Sym	"$1 $2"	
1.0	H	Func	->	Let	2Let	"\mathop{$1$2}"	
1.0	H	Func	->	2Let	2Let	"\mathop{$1$2}"	
1.0	H	2Let	->	Let	Let	"$1$2"	
1.0	V	Exp	->	Func	Exp	"$1_{$2}"	BBB
1.0	V	Exp	->	Func	Sym	"$1_{$2}"	BBB

Terminal production rules are not shown in Table 3.2 because there are large numbers of terminals. Non-terminals *Sym*, *Let*, *Over*, *BigOp*, *OpUn*, *ROpUn*, *OpBin*, *OverSym*, *SupSym*, *LeftPar*, *RightPar* and *Sqrt* have terminal production rules. For example, BigOp includes big operators like \sum (∑), \bigcup (∪), \cap (∩), \int (∫),

\prod (∏), \lim (lim). *OpUn* includes unary operators like +, -, \neg (≠), \pmv(±), \log (log), \sin (sin), \cos (cos), \tan (tan), \exists (∃), \forall (∀), \ldots (⋯).

3.2 Segmentation Hypothesis Generator

In the case of on-line handwritten recognition, the input is a set of strokes (shown as Figure 3.2). As we mentioned before, the segmentation hypothesis generator is to segment the mathematical expression into groups which are called segmentation hypotheses. These segmentation hypotheses will be recognized by a classifier and each of them is assumed to be a single symbol.




```
<trace id="0">
1.03091 6.55171, 1.03134 6.55171, 1.03176 6.55171, 1.03262 6.55171, 1.03432 6.54915,
1.03731 6.54532, 1.04072 6.54105, 1.04371 6.53721, 1.04669 6.53167, 1.04925 6.52442,
1.05138 6.51717, 1.05224 6.51034, 1.05309 6.50565, 1.05394 6.50138, 1.05394 6.50053,
1.05394 6.5001, 1.05394 6.50053, 1.05394 6.50096, 1.05394 6.50181, 1.05437 6.50224,
1.05437 6.50394, 1.05522 6.50608, 1.05693 6.50949, 1.05863 6.51589, 1.06034 6.52399,
1.06205 6.53167, 1.06375 6.54062, 1.06631 6.54873, 1.06802 6.55342, 1.06887 6.5547,
1.06887 6.55513, 1.06845 6.55513, 1.06845 6.5547, 1.06802 6.55427
</trace>
<trace id="1">
1.0356 6.53977, 1.03646 6.53934, 1.03816 6.53806, 1.04371 6.53593, 1.05053 6.5338,
1.05608 6.53337, 1.05991 6.53295, 1.06247 6.53337
</trace>
<trace id="2">
1.1047 6.53337, 1.10513 6.53295, 1.10641 6.53252, 1.10769 6.53209, 1.10982 6.53209,
1.11366 6.53124, 1.1192 6.52953, 1.12432 6.52868, 1.12944 6.52783, 1.13157 6.52911
</trace>
    ⋮
<trace id="8">
1.30943 6.50949, 1.30901 6.50949, 1.30901 6.50906, 1.30858 6.50906, 1.30773 6.50863,
1.30687 6.50863, 1.30431 6.50906, 1.30175 6.50949, 1.2992 6.51034, 1.29664 6.51077,
1.2928 6.51461, 1.29024 6.51802, 1.28811 6.52356, 1.2864 6.53167, 1.28725 6.53934,
1.28981 6.54276, 1.29365 6.54574, 1.29962 6.54617, 1.30773 6.54489, 1.3154 6.54318,
1.32009 6.5419, 1.32095 6.54148, 1.3218 6.54148, 1.32137 6.54148, 1.32265 6.54105
</trace>
```

Figure 3.2 The input in on-line case is a set of strokes

The goal of segmentation hypothesis generator is to find out the correct segmentation as many as possible. Since many symbols are composed by only one stroke, in our system, each single stroke will be considered to be a hypothesis.

However, it is not enough because there are still a large number of multi-stroke symbols which are made of more than one stroke. For example, symbols like “i”, “j”, “=” are composed by two strokes. Symbols like “≠”, “÷”, “⋯” are composed by three strokes. Symbols like “sin”, “cos”, “lim” are composed by four strokes. As a result, not only each single stroke but also multiple strokes should be taken into account in the segmentation hypothesis generator.

A possible way to treat this problem is by merging closer strokes to form a hypothesis. To measure the closeness between two strokes, first of all, we use minimum bounding box. The minimum bounding box is a term used in geometry. For a point set *S*, the minimum bounding box refers to the rectangle box with the smallest area within which all the points lie. As is shown in Figure 3.3, the dash lines represent the bounding box for each stroke in the expression.

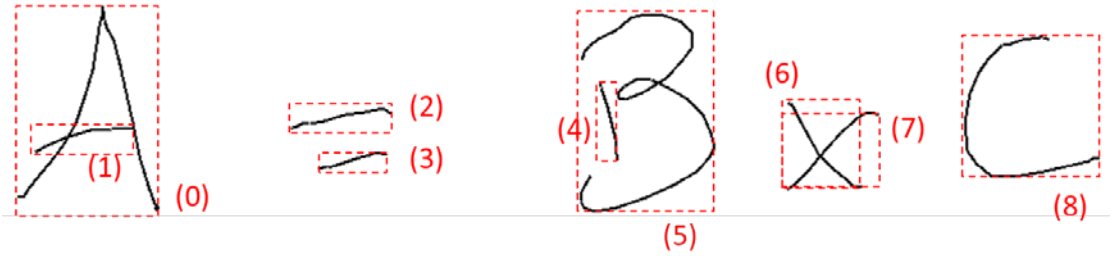


Figure 3.3 Bounding box of each stroke in the expression

Then, we also define a searching area for the stroke. Take stroke 0 in Figure 3.3 for example. As is shown in Figure 3.4, the bounding box of this stroke is the rectangle labeled by B. To represent a rectangle area in our paper, we would use $\{(x,y),(s,t)\}$, where (x,y) is top left corner and (s,t) is bottom right corner. Bounding box B can be denoted by $\{(x,y),(s,t)\}$. The searching area labeled by A is given relative to the position and size of bounding box B. The searching area A can be represented by $\{(x-RX,y-RY),(s+RX,t+RY)\}$ where $(x-RX,y-RY)$ is top left corner, $(s+RX,t+RY)$ is bottom right corner, RX and RY are called reference distance.

Let w_i and h_i be the width and height of the bounding box of the i^{th} stroke in an expression ($\mathrm{i}=1\cdots\mathrm{n}$; n is the number of strokes in the expression). Let $W=\{w_1,w_2,\cdots,w_n\}$ and $H=\{h_1,h_2,\cdots,h_n\}$ be the set of width and height of bounding box of all the strokes. RX and RY are computed as follow:

$$RX=max\big(mean(W),median(W)\big)$$

$$RY=max\big(mean(H),median(H)\big)$$

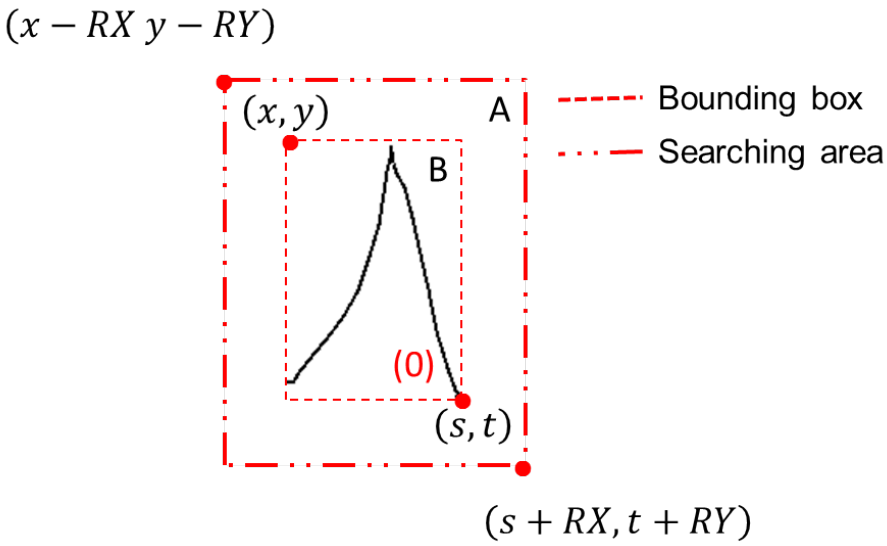
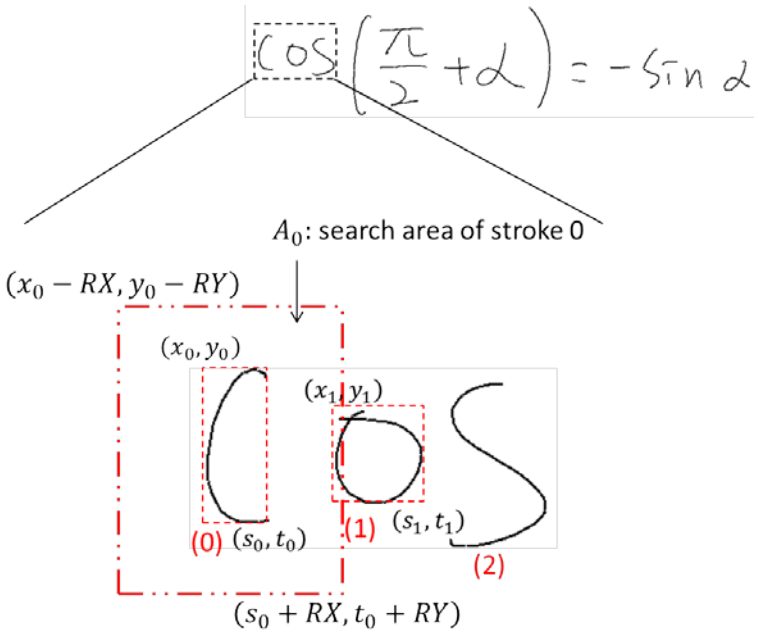


Figure 3.4 Searching area of a stroke

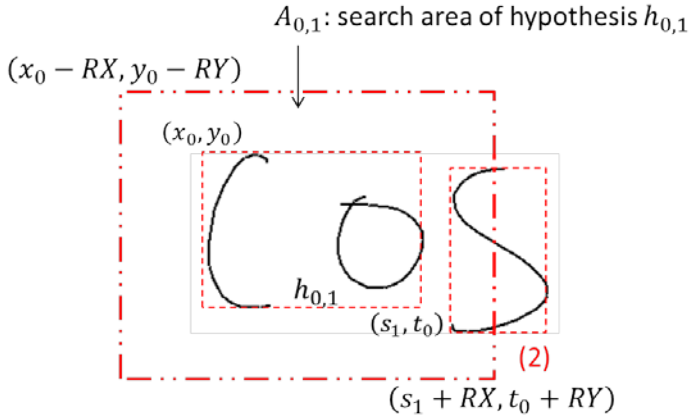
Given a certain stroke s and its searching area A_s, any other stroke is considered to be close to stroke s only if its bounding box has overlap with searching area A_s. Take stroke 0 in Figure 3.5 for example, A_0 is searching area of stroke 0. Stroke 1 is close to stroke 0 because it is in the searching area A_0. As a result, stroke 0 and stroke 1 form a hypothesis and it will be recognized by classifier in the following step. The generator inevitably generates wrong segmentation. Take stroke 5 for another example, stroke 5 and stroke 4 combine to be a hypothesis in a similar way. There are another two hypotheses (stroke 5 and 6, stroke 5 and 7) because the bounding box of stroke 6 and 7 have overlap with searching area A_5. But obviously, they are not a symbol and not a correct segmentation. The goal of segmentation hypothesis generator is to find out the correct segmentation as many as possible. Larger searching area generates more hypotheses and covers more correct segmentation. But it introduces too much wrong segmentation and increases the hypotheses space. To solve this problem, we can set a "junk class" in the classifying step to avoid improbable hypothesis. This will be introduced in the next section. We can also apply machine learning to obtain an optimal searching area from a training set.

Figure 3.5 Search for closer strokes using searching area

The method introduced above is to generate two-stroke hypothesis. To generate hypotheses which are composed by three strokes, we just need to merge a two-stroke hypothesis with one of its closer strokes to form a three-stroke hypothesis. It is similar to form four-stroke hypothesis, five-stroke hypothesis and even n-stroke hypothesis. Take symbol "cos" for example. In Figure 3.6(a), stroke 0 and stroke 1 form a two-stroke hypothesis $h_{0,1}$. To find any other strokes close to $h_{0,1}$, we need the searching area for this two-stroke hypothesis. The searching area and bounding box for n-stroke case ($n \geq 2$) is similar with one-stroke case. The bounding box of stroke 0 is $\{(x_0, y_0), (s_0, t_0)\}$ and the bounding box of stroke 1 is $\{(x_1, y_1), (s_1, t_1)\}$. The bounding box of $h_{0,1}$ is the rectangle area $\{(x_0, y_0), (s_1, t_0)\}$, within which all the points of stroke 0 and 1 lie. The searching area $A_{0,1}$ is $\{(x_0 - RX, y_0 - RY), (s_1 + RX, t_0 + RY)\}$. Any other stroke is considered to be close to $h_{0,1}$ only if its bounding box has overlap with searching area $A_{0,1}$. In Figure 3.6(b), stroke 2 is close to $h_{0,1}$ because its bounding box has overlap with searching area $A_{0,1}$. Thus, $h_{0,1}$ and stroke 2 form a three-stroke hypothesis and it will be recognized by classifier in the following step.



(a)



(b)

Figure 3.6 Generate three-stroke hypothesis. (a) Merge stroke 0 and stroke 1 to form a two-stroke hypothesis. (b) Merge two-stroke hypothesis in (a) and stroke 2 to form a three-stroke hypothesis.

To form three-stroke hypothesis, we need to find out all two-stroke hypothesis. Similarly, we need three-stroke hypotheses to form four-stroke hypotheses. And so on, we need (n-1) stroke hypotheses to form n-stroke hypotheses ($n \geq 2$). It is a recursive way. In our system, we generate up to four-stroke hypotheses because it covers all the cases in our experiment dataset. It is important to note that it also introduces wrong segmentation in this way. Take "cos" for example. To form a three-stroke hypothesis of "cos", we need to firstly form two-stroke hypotheses of "co" or "os". But these two-stroke hypotheses are just wrong segmentation and they increase the hypotheses space. As is said before, a junk class in the classifier can avoid improbable hypothesis and help to solve this problem.

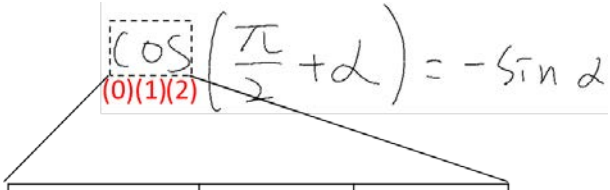
3.3 Symbol Recognition

A set of segmentation hypotheses is obtained as described in section 3.2. Then, the symbol classifier associates a recognition probability and a class label with each segmentation hypothesis. In our system, Multilayer Perceptron Neural Network (MLP) was chosen. For a given hypothesis h_i, MLP gives it a probability denoted by $p(c_j|h_i)$ with $\sum_j p(c_j|h_i) = 1$. $p(c_j|h_i)$ represents the probability that this hypothesis h_i being the class c_j. For every hypothesis, the classifier gives them many symbol candidates. It is not necessary to keep all the candidates because many ambiguities in classifying stage could be resolved at the global context level. After experiments, we found that keeping the best three candidates of the symbol classifier is already enough.

As is introduced in Section 3.2, the segmentation hypothesis generator always generates wrong segmentation and consequently increases hypotheses space. Thus,

we hope the classifier can identify the wrong segmentation and reject them. In a word, we are dealing with rejection problem. In this paper, we will call the reject class the "junk class". The "junk class" is also a class label, but it doesn't represent any symbol. For a hypothesis h_i being recognized as junk, the classifier also gives it a probability $p(c_i = junk|h_i)$. The higher the probability is, the more probably it can be a wrong segmentation. The junk class is like a filter. It rejects wrong segmentation, and consequently limits the hypotheses space.

Figure 3.7 shows the classifying result for symbol "cos". As is described in Section 3.2, to form a three-stroke hypothesis $h_{0,1,2}$ of "cos", we need to firstly form two-stroke hypothesis $h_{0,1}$ of "co" and $h_{1,2}$ of "os". Hypotheses $h_{0,1}$and $h_{1,2}$are wrong segmentation. Both of them are considered to be junk with high probability. Thus, they are rejected from the following CYK parsing step.



Segmentation hypothesis	Class label	Probability
$h_{0,1}$	junk	0.945029
	0	0.023312
	x	0.019574
$h_{1,2}$	junk	0.992379
	\sqrt (√)	0.004120
	1	0.001873
$h_{0,1,2}$	\cos	0.862543
	junk	0.136511
	a	0.000411

Figure 3.7 Classifying result for symbol "cos"

3.4 CYK Parsing

After segmentation and symbol recognition, a set of hypotheses associated with class label and probability is obtained. We need to determine the spatial relation among these symbols in order to build a complete structure. In our work, the CYK parsing algorithm is used to parse the input (represented by the set of hypotheses) and obtain the most probable derivation. The CYK algorithm is a dynamic programming method, and based on the construction of a parsing table.

Let $\mathcal{G}$ be a CNF 2D-SCFG. As is introduced before, the probabilities are formally defined as:

$$p(A \to t)$$

$$p(A \xrightarrow{spr} BC)$$

$$\mathrm{A, B, C} \in V_N, t \in V_T$$

Let $\mathcal{S} = \{s_i | i: 1,2, \dots N\}$ be the set of all the strokes in a given expression where N is the total number of strokes. Let S_l be the set of l strokes $(1 \leq l \leq N)$ and $S_l \subseteq \mathcal{S}$. Let $\mathcal{T}$ be the parsing table of CYK algorithm. Each element in table $\mathcal{T}$ is denoted by $e_l(A, S_l, p)$ and defined as follow. For a given element $e_l(A, S_l, p)$, p represents the probability that A is the solution of the mathematical sub-expression composed by the l strokes S_l. Let $T_l = \{e_l(A, S_l, p)\}$ be the parse structure where each element $e_l(A, S_l, p)$ is composed by l strokes.

The CYK algorithm is to calculate the parsing table $\mathcal{T}$ (see Figure 3.8). Each element $e_l(A, S_l, p)$ represents a sub-expression composed by l strokes. Each cell is the set of elements with the same strokes S_l. Each row is the set of cells with the same number of strokes, that is $T_l = \{e_l(A, S_l, p)\}$. The table begins at the bottom row T_1 where each element $e_1(A, S_1, p)$ is composed by only one stroke. The algorithm constructs the higher part of table by calculating new sub-expression of increasing strokes. The top of table $T_N = \{e_N(A, S_N, p)\} = \{e_N(A, \mathcal{S}, p)\}$ is the elements composed by all the strokes of a given expression. In the element $e_N(A, \mathcal{S}, p)$ with the highest probability p, A is the most probable interpretation for this expression.

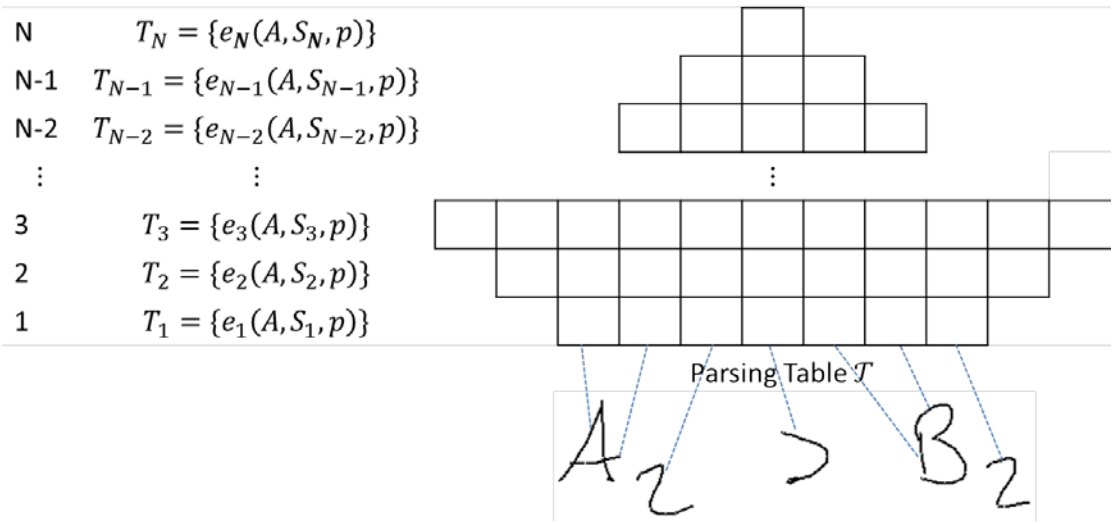


Figure 3.8 CYK parsing table

The CYK algorithm is divided into two steps. First it initializes the table with the segmentation hypotheses. Let h_l be the segmentation hypothesis composed by l strokes $(l \leq 4)$. $l \leq 4$ because we generate up to four-stroke hypotheses as is described in section 3.2. The initialization works as follow:

$$T_1 = T_1 \cup \{e_1(A, h_1, p)\}$$
$$T_2 = T_2 \cup \{e_2(A, h_2, p)\}$$
$$T_3 = T_3 \cup \{e_3(A, h_3, p)\}$$
$$T_4 = T_4 \cup \{e_4(A, h_4, p)\}$$

Generally,

$$T_l = T_l \cup \{e_l(A, h_l, p)\} \qquad l = 1,2,3,4$$

For a given $e_l(A, h_l, p)$, the value p is:

$$p_l(A) = p(A \to t)p(t|h_l)$$

where t is a particular mathematical symbol, A is a nonterminal and $A \to t$ is a terminal production. Value $p(A \to t)$ is the probability provided by terminal production $A \to t$ in the stochastic context-free grammar, and $p(t|h_l)$ is the probability that this hypothesis h_l being recognized as the class t. $p(t|h_l)$ is provided by MLP symbol classifier (see Section 3.3).

It is important to note that one mathematical symbol t can belong to several nonterminal A. For example, the "+" symbol can be a binary operator such as "1+2", or it can be a unary operator such as "+2". The "-" symbol can be a binary operator such as "2-1", or a unary operator such as "-2", or a fractional line such as "$\frac{1}{2}$". Thus for a hypothesis h_l being classified as a specific class t, each possible nonterminal associated to the corresponding rule is added to the parsing table. For example, "+" is added in the table with its probability for being "OpBin", "OpUn". "-" is added in the table with its probability for being "OpBin", "OpUn" and "Over".

The pseudocode for the initialization is shown in Figure 3.9. For each hypothesis, MLP classifier gives the best three candidates (see Section 3.3). We use a threshold to avoid exploring improbable hypothesis. This is done to limit the search space during the parsing. But in this way, we could not guarantee that the optimal solution is achieved because some possible solutions are removed by threshold.

```
For all segmentation hypotheses h_l do {
        for all nonterminals A of terminal productions (A → t) do {
                p(t|h_l) = MLP(h_l)                    //Symbol Classifier
                p = p(A → t)p(t|h_l)
                if (p>threshold) then
                        T_l = T_l ∪ {e_l(A, h_l, p)}
        }
}
```

Figure 3.9 Pseudocode for CYK parsing table initialization

Then, the parsing process constructs the higher part of table by calculating new subexpression of increasing size. This step is computed as:

$$T_l = T_l \cup \{e_l(A, S_l, p)\} \qquad l = 2,3,\cdots,N$$

A new subexpression $e_l(A, S_l, p)$ is created from two subexpression of smaller size $e_k(B, S_k, p_k)$ and $e_{l-k}(C, S_{l-k}, p_{l-k})$ $(1 \le k < l)$ according to both syntactic constraint (the production rule $A \xrightarrow{spr} BC$) and the spatial constraint (spatial relation spr). For a given $e_l(A, S_l, p)$, the value p is defined as:

$$p = p(A \xrightarrow{spr} BC)p_k p_{l-k} p(S_k, S_{l-k}|spr)$$

where $p(A \xrightarrow{spr} BC)$ is the probability provided by production rule $A \xrightarrow{spr} BC$ in the grammar, the probability p_k and p_{l-k} are obtained from the lower part of CYK parsing table, and $p(S_k, S_{l-k}|spr)$ is the probability that these two sets of strokes S_k, S_{l-k} are combined according to the spatial relation spr (see Section 3.5).

When creating a subexpression of size l, a straightforward way is to try all $(k, l-k)$ size pairs. In fact, it is not necessary to check all combinations. For a given subexpression a of size k, we define a specific searching area according to different kinds of spatial relation (see Figure 3.10). The size of searching area (dashed line area) is given by the bounding box of a and reference distance RX, RY (see Section 3.2). Any other subexpression b of size $l-k$, which has overlap with the searching area, will be combined with a to form a new subexpression of size l. Figure 3.11 shows two cases where given a subexpression ("sin" and "2") and their searching area (dashed line area). Given the subexpression "sin" and the spatial relation "horizontal", the system only would apply the horizontal production rules with the subexpressions which overlap the searching area. The denominator "2" is other example of space search for "vertical up" subexpressions.

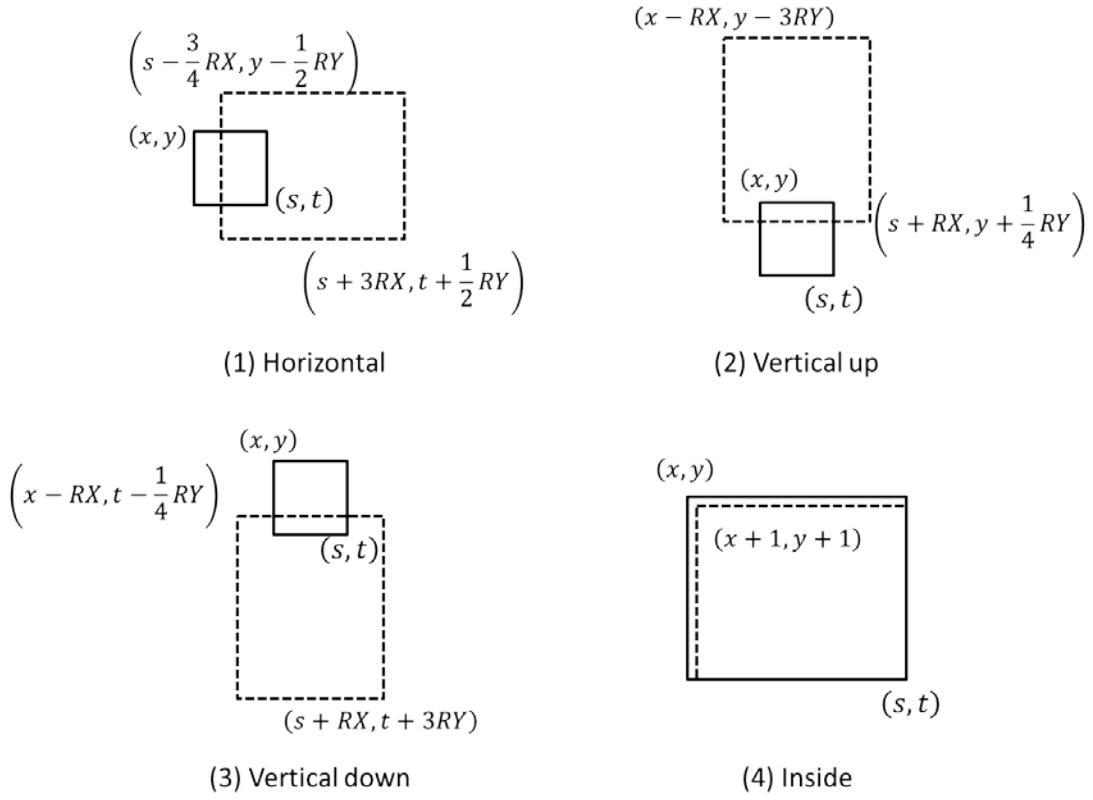


Figure 3.10 Searching area for different kinds of spatial relation

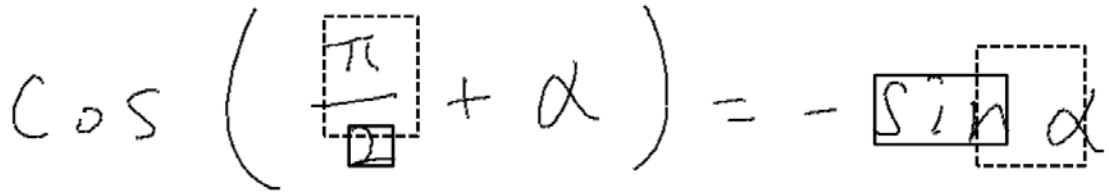


Figure 3.11 Example of searching area for a particular subexpression and spatial relation

Figure 3.12 shows the second step of CYK parsing algorithm. In this pseudocode, the "search" operation is to search subexpressions with searching area as explained above.

```
For all l = 2,3,⋯,N do {   // N strokes
    for all k = 1,2,⋯,l − 1 do {
      for all e_k(B, S_k, p_k) ∈ L_k do {
            z_h = search(T_{l−k}, S_k, horizontal)
            z_v = search(T_{l−k}, S_k, vertical)
            z_i = search(T_{l−k}, S_k, inside)
            for all e_{l−k}(C, S_{l−k}, p_{l−k}) ∈ z_h do {
                for all (A ⟶^{spr} BC) such that spr ∈ {Sub, Sup, H, SSE} do {
                    p = p(A ⟶^{spr} BC) p_k p_{l−k} p(S_k, S_{l−k} | spr)
                    if (p > 0.0) then
                        T_l = T_l ∪ {e_l(A, S_k ∪ S_{l−k}, p)}
                }
            }
            for all e_{l−k}(C, S_{l−k}, p_{l−k}) ∈ z_v do {
                for all (A ⟶^{spr} BC) such that spr ∈ {V} do {
                    p = p(A ⟶^{spr} BC) p_k p_{l−k} p(S_k, S_{l−k} | spr)
                    if (p > 0.0) then
                        T_l = T_l ∪ {e_l(A, S_k ∪ S_{l−k}, p)}
                }
            }
            for all e_{l−k}(C, S_{l−k}, p_{l−k}) ∈ z_i do {
                for all (A ⟶^{spr} BC) such that spr ∈ {Ins} do {
                    p = p(A ⟶^{spr} BC) p_k p_{l−k} p(S_k, S_{l−k} | spr)
                    if (p > 0.0) then
                        T_l = T_l ∪ {e_l(A, S_k ∪ S_{l−k}, p)}
                }
            }
        }
    }
}
```

Figure 3.12 Pseudocode of the CYK parsing algorithm

Now we compare our 2D CYK algorithm with the 1D case. In the standard 1D CYK algorithm, two indexes explain the positions that define some substring. In the 2D CYK algorithm, there is only one index. There is a level for each subexpression size. Each level stores a set of elements in the same size. In the initialization step, the terminals are added at T_1, T_2, T_3, T_4. After that, the parsing process continues by creating new subexpressions of increasing size, in which both syntactic constraints (grammar) and spatial constraints are taken into account for each new subexpression.

3.5 Spatial Relation

In our system, we define six kinds of spatial relation (see Figure 3.13): subscript (Sub), superscript (Sup), horizontal (H), sub-super-expression (SSE), vertical (V) and inside (Ins).

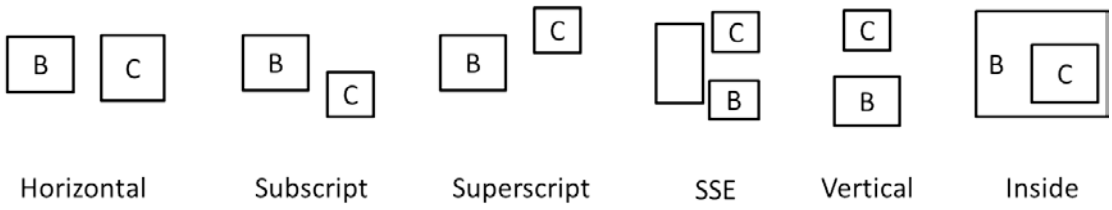


Figure 3.13 Spatial relation between two subexpressions B and C

Given two subexpressions B and C, the spatial relation between them is determined using some geometric features of their bounding boxes. Using these features, some functions are defined by us to compute the probability $p(S_k, S_{l-k}|spr)$ used in the CYK parsing algorithm.

Before introducing the features and functions, we defined six reference lines (see Figure 3.14(a)) on the bounding box: left sub-line, left centroid-line, left sup-line and right sub-line , right centroid-line , and right sup-line (denoted by $lSub$, $lCen$, $lSup$, $rSub$, $rCen$, and $rSup$ respectively). These reference lines are used to extract geometric features of the bounding box. Firstly, three types of symbols are defined by us: normal, ascending and descending. For example, “a”, “c”, “e” are normal symbol, “b”, “d” are ascending symbol, “p”, “y”, “q” are descending symbol. The reference lines are adapted for different types of symbol. In the symbol recognition step, the reference lines are modified according to this classification.

For normal symbol, the centroid is the geometric center (see Figure 3.14(b) left):

$$lCen = rCen = \frac{y + t}{2}$$

$$lSub = rSub = rCen + 0.9 \cdot (t - rCen)$$

$$lSup = rSup = y + 0.1 \cdot (rCen - y)$$

For ascending symbol, the centroid is displaced down to $(centroid + bottom)/2$ (see Figure 3.14(b) middle):

$$lCen = rCen = \frac{\frac{y+t}{2} + t}{2}$$

$$lSub = rSub = rCen + 0.9 \cdot (t - rCen)$$

$$lSup = rSup = rCen + \frac{y + rCen}{2}$$

For descending symbol, the centroid is displaced up to $(centroid + top)/2$ (see Figure 3.14(b) right):

$$lCen = rCen = \frac{y + \frac{y+t}{2}}{2}$$

$$lSub = rSub = rCen + \frac{rCen + t}{2}$$

$$lSup = rSup = y + 0.1 \cdot (rCen - y)$$

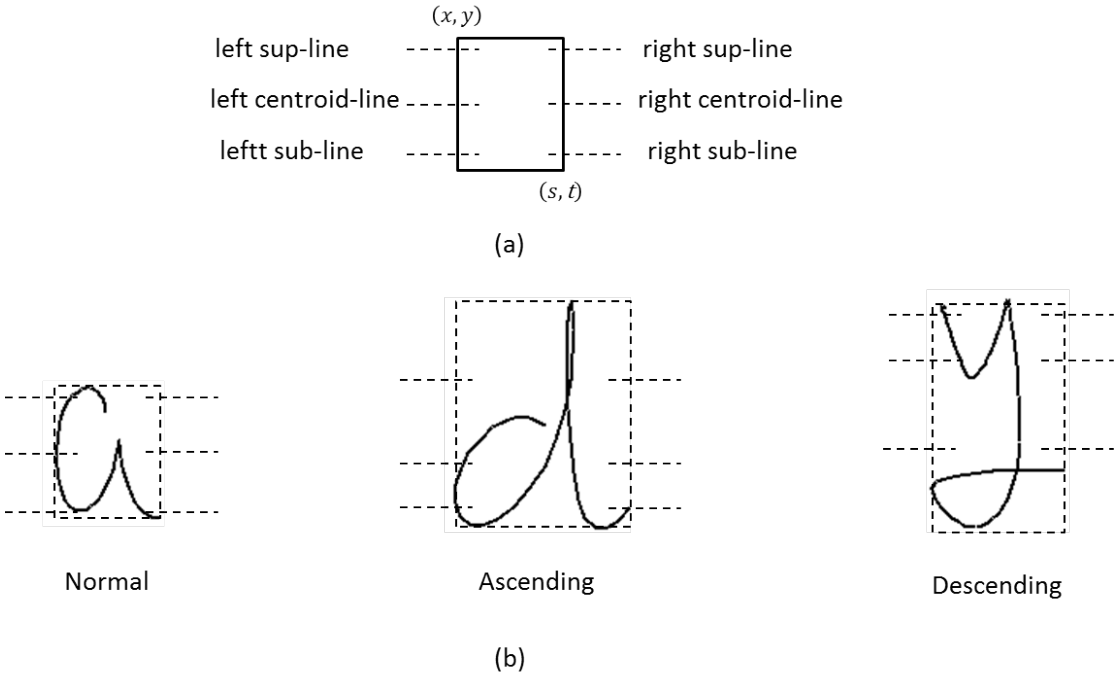


Figure 3.14 (a) Reference line for a general case (b) Reference line for different kinds of symbol: normal, ascending and descending

Once the reference lines are calculated for every single symbol, this information is hierarchical inherited as follows. The combination of two subexpressions B and C resulting in a new subexpression A should follow some rules in order to preserve good reference lines. The reference line of A is different according to different spatial relation.

For horizontal relation ($B\ C$) (see Figure 3.15),

$$lSup(A) = \frac{lSup(B) + lSup(C)}{2}, \qquad rSup(A) = \frac{rSup(B) + rSup(C)}{2}$$

$$lCen(A) = \frac{lCen(B) + lCen(C)}{2}, \qquad rCen(A) = \frac{rCen(B) + rCen(C)}{2}$$
$$lSub(A) = \frac{lSub(B) + lSub(C)}{2}, \qquad rSub(A) = \frac{rSub(B) + rSub(C)}{2}$$

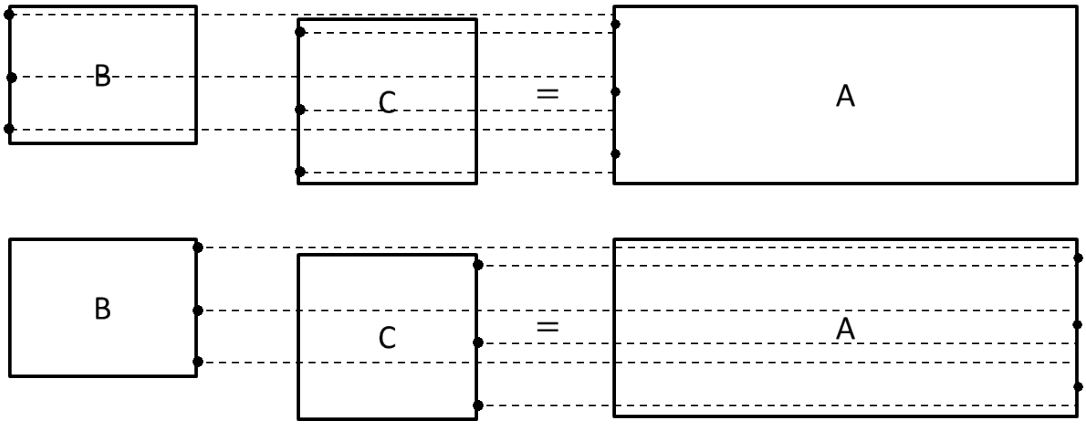


Figure 3.15 Reference line for horizontal relation

For subscript relation (B_C) (see Figure 3.16),

$$lSup(A) = lSup(B), \qquad rSup(A) = rSup(B)$$
$$lCen(A) = lCen(B), \qquad rCen(A) = rCen(B)$$
$$lSub(A) = lSub(B), \qquad rSub(A) = rSub(C)$$

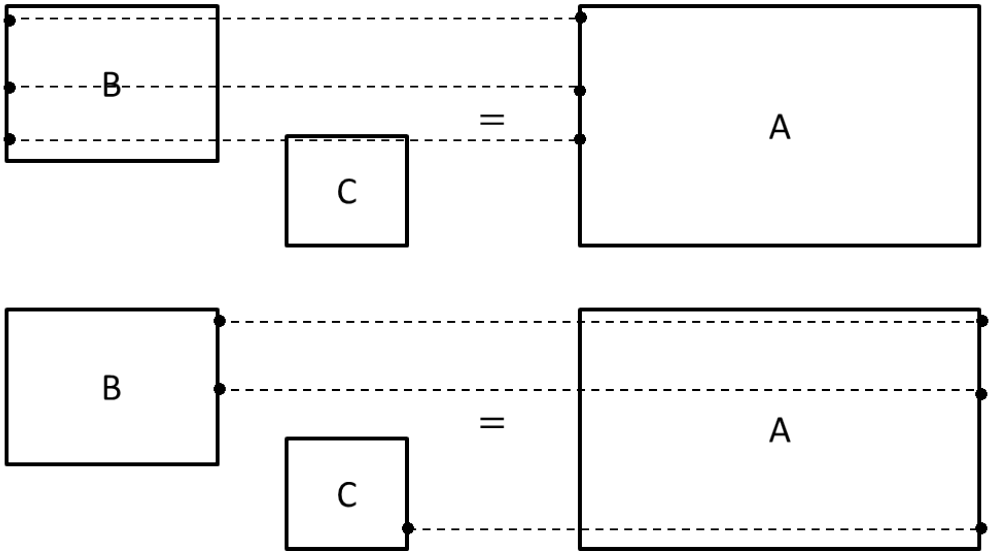


Figure 3.16 Reference line for subscript relation

For superscript relation (B^C) (see Figure 3.17),

$$lSup(A) = lSup(B), \qquad rSup(A) = rSup(C)$$
$$lCen(A) = lCen(B), \qquad rCen(A) = rCen(B)$$
$$lSub(A) = lSub(B), \qquad rSub(A) = rSub(B)$$

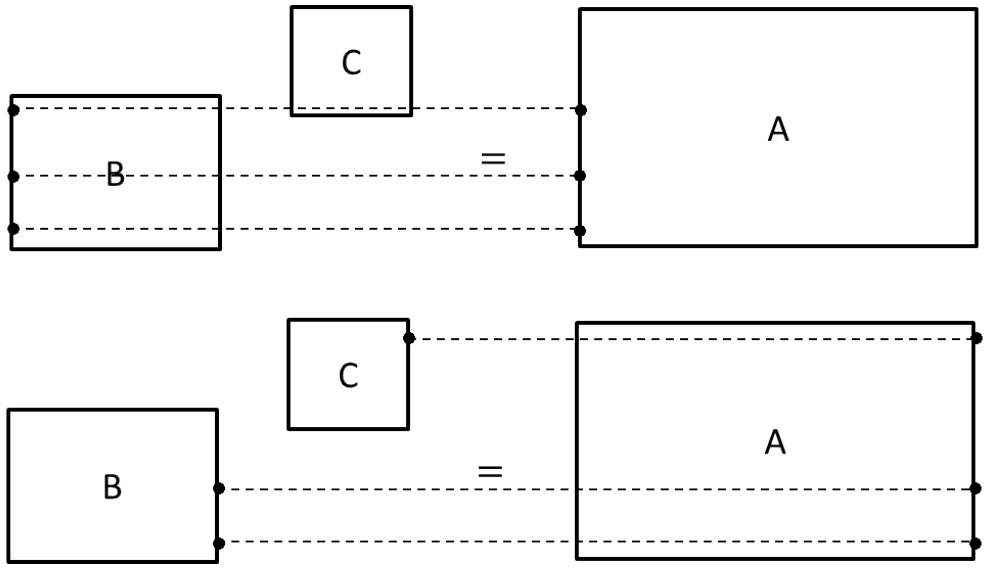


Figure 3.17 Reference line for superscript relation

For sub-super-expression (X_C^B) (see Figure 3.18),

$$lSup(A) = lSup(B), \qquad rSup(A) = rSup(B)$$

$$lCen(A) = \frac{lCen(B) + lCen(C)}{2}, \qquad rCen(A) = \frac{rCen(B) + rCen(C)}{2}$$

$$lSub(A) = lSub(C), \qquad rSub(A) = rSub(C)$$

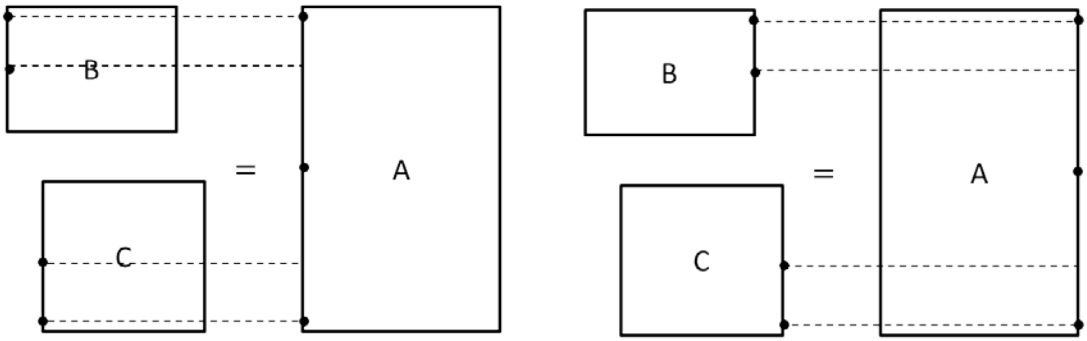


Figure 3.18 Reference line for sub-super-expression

For inside relation ($\sqrt{C}$) (see Figure 3.19),

$$lSup(A) = lSup(B), \qquad rSup(A) = \frac{rSup(B) + rSup(C)}{2}$$

$$lCen(A) = LCen(B), \qquad rCen(A) = \frac{rCen(B) + rCen(C)}{2}$$

$$lSub(A) = lSub(B), \qquad rSub(A) = \frac{rSub(B) + rSub(C)}{2}$$

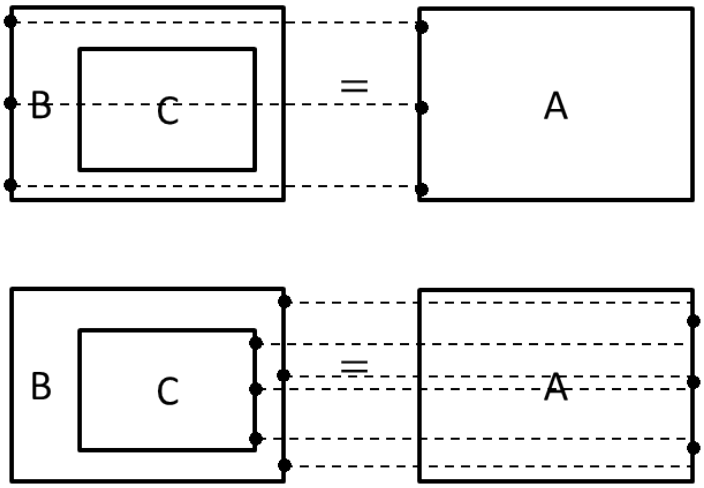


Figure 3.19 Reference line for inside relation

The vertical relation ($\binom{B}{C}$) depend on the grammar. As is described in Section 3.1, there is an additional merge flag in the production rules of vertical relation. For example, in the production rule

1.0 V Exp -> Exp OverExp "\frac{$1}{$2}" BCC

Merge flag "BCC" represents that father (*Exp*) uses sup-line of A (*Exp*), and center-line and sub-line of B (*OverExp*).

Finally, the features and functions for determining the relation between B and C are explained as below:

(1) Horizontal (see Figure 3.20): the features are horizontal distance (denoted by dx) and difference between $rCen(B)$ and $lCen(C)$ (denoted by dvc). The probability function is computed as:

$$p(S_k, S_{l-k}|H) = \frac{p_1 + p_2}{2}$$

such that:

$$p_1 = 1 - \frac{dx}{3 \cdot RX}$$

$$p_2 = 1 - \frac{dvc}{\max(RY, h_B)}$$

where h_B is height of bounding box of B, RX and RY are reference distance.

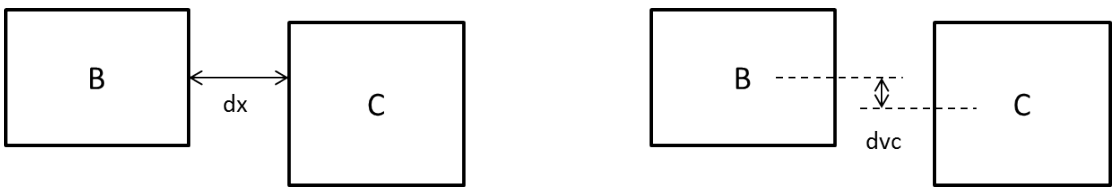


Figure 3.20 Features for horizontal relation

(2) Subscript (see Figure 3.21): the features are horizontal distance (denoted by dx)

and difference between $rSub(B)$ and $lCen(C)$ (denoted by dy). The probability function is computed as:

$$p(S_k, S_{l-k}|sub) = \frac{p_1 + p_2}{2}$$

such that:

$$p_1 = 1 - \frac{dx}{3 \cdot RX}$$

$$p_2 = 1 - \frac{dy}{\max(RY, h_B)}$$

where h_B is height of bounding box of B, RX and RY are reference distance.

Figure 3.21 Features for subscript relation

(3) Superscript (see Figure 3.22): the features are horizontal distance (denoted by dx) and difference between $rSup(B)$ and $lCen(C)$ (denoted by dy). The probability function is computed as:

$$p(S_k, S_{l-k}|sup) = \frac{p_1 + p_2}{2}$$

such that:

$$p_1 = 1 - \frac{dx}{3 \cdot RX}$$

$$p_2 = 1 - \frac{dy}{\max(RY, h_B)}$$

where h_B is height of bounding box of B, RX and RY are reference distance.

Figure 3.22 Features for superscript relation

(4) Vertical (see Figure 3.23): the features are vertical distance (denoted by dy), difference between the horizontal centers (denoted by dhc), difference between left boundary dl, difference between right boundary dr. The probability function

is computed as:

$$p(S_k, S_{l-k}|V) = \frac{p_1 + p_2}{2}$$

such that:

$$p_1 = 1 - \frac{dy}{3 \cdot RY}$$

$$p_2 = 1 - \frac{dl + dr}{3 \cdot RX}$$

where RX and RY are reference distance.

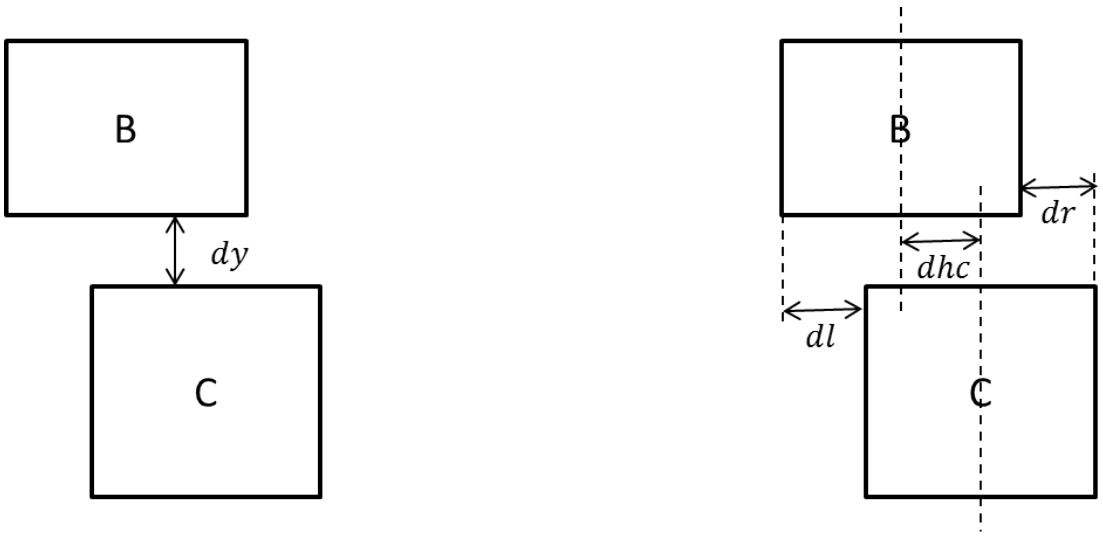


Figure 3.23 Features for vertical relation

(5) Sub-super-expression (see Figure 3.24): the features are vertical distance (denoted by dy) and the difference between left boundary dl. The probability function is computed as:

$$p(S_k, S_{l-k}|SSE) = \frac{p_1 + p_2}{2}$$

such that:

$$p_1 = 1 - \frac{dy}{3 \cdot RY}$$

$$p_2 = 1 - \frac{dl}{3 \cdot RX}$$

where RX and RY are reference distance.

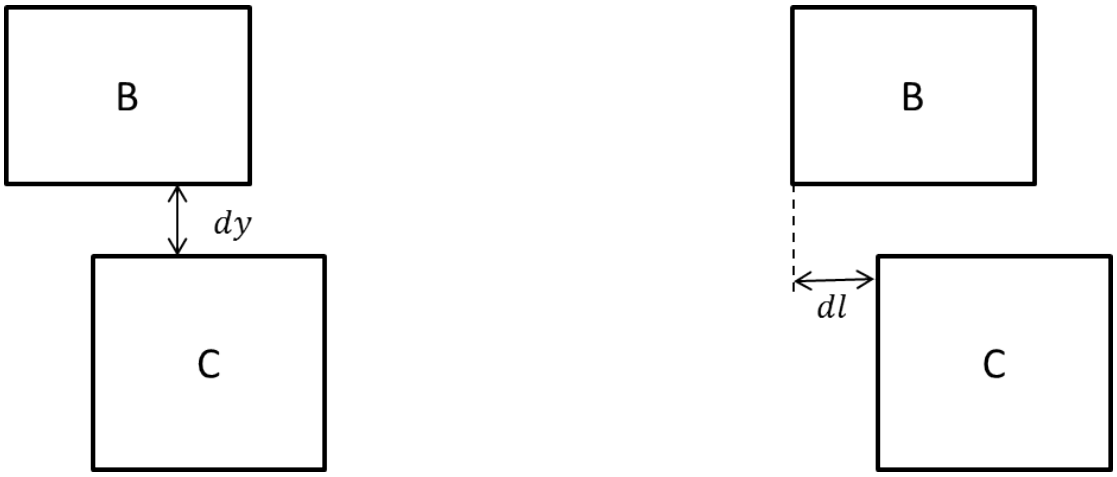


Figure 3.24 Features for sub-super-expression relation

(6) Inside (see Figure 3.25): the features are the horizontal distance (denoted by dx) and vertical distance (denoted by dy). The probability function is computed as:

$$p(S_k, S_{l-k}|Ins) = 1 - \frac{(dx)^2 + (dy)^2}{(RX)^2 + (RY)^2}$$

where RX and RY are reference distance.

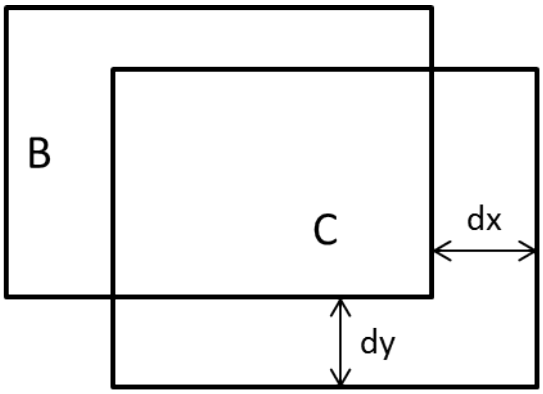


Figure 3.25 Features for inside relation

3.6 Parsing Output

After the CYK parsing algorithm is performed, the recognized expression can be obtained by going through the parsing tree that covers the root with both initial state and highest probability. The parsing result could be presented in different ways such as MathML or LaTex. In our system, it produces the output in both MathML and LaTex format.

The process that generates the desired format output from the parsing tree is just a recursive way from the root to the leaves. Sometimes, the mathematical expression is not fully recognized. But it is still meaningful to provide and output part of the expression. For example (see Figure 3.26), the input expression is not fully

recognized because the square root is missed. However, the partial output is still useful. In the case of partial recognizing, the system looks for the most probable subexpression of bigger size that covers the initial symbol (Exp or Sym) of the grammar. Then it goes through from this node to the leaves to generate the output.

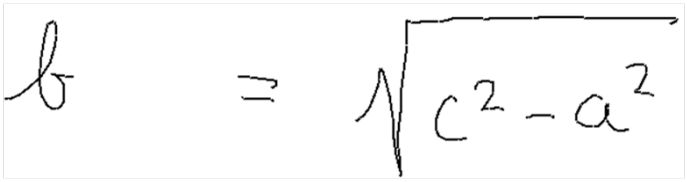
Input	Latex Output
	b = {C}^{2} - {a}^{2}

Figure 3.26 Example of partial recognizing

3.7 Example

In order to illustrate our system, we present a simple example (see Figure 3.27). First of all, the segmentation hypothesis generator obtains 10 hypotheses including 7 one-stroke hypotheses, 2 two-stroke hypotheses and 1 three-stroke hypotheses. Then, the symbol classifier (MLP) associates a recognition probability and a class label with each segmentation hypothesis. All of these information are added to T_1, T_2, T_3 to initialize the parsing table (see Figure 3.28). As is described in Section 3.4, one mathematical symbol can belong to several nonterminals. For example, for stroke 0, "-" is added in the table with its probability for being "OpBin", "OpUn" and "Over". For stroke 2,3, "k" is added in the table with its probability for being "Sym" and "Let".

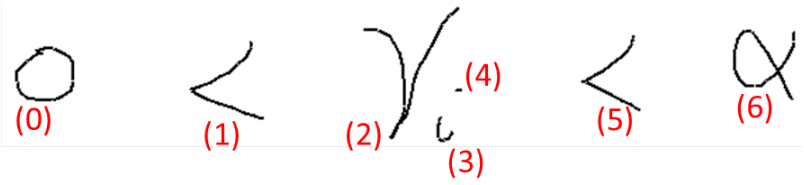


Figure 3.27 Sample expression

Once the table is initialized, the recursive step begins to build subexpression until it completes subexpression of size 7 (all the input strokes). When building new subexpression, both syntactic constraints (grammar defined by us, see Section 3.1) and spatial constraints (see Section 3.5) are taken into account. Figure 3.29 shows the complete parsing table for the sample expression.

After the CYK parsing algorithm is performed, a parsing tree can be obtained from the table. At the top of table, there are two possible solutions Exp and $OBExp$. Only initial state Exp or Sym can become the root of parsing tree. Thus, the nonterminal Exp with the probability -3.50054 is chosen. Once the root is determined, the parsing tree can be obtained by going through from top to bottom

(see Figure 3.30(a)). The parsing tree is shown as Figure 3.30(b). Finally, the recognized expression can be obtained by going through the parsing tree.

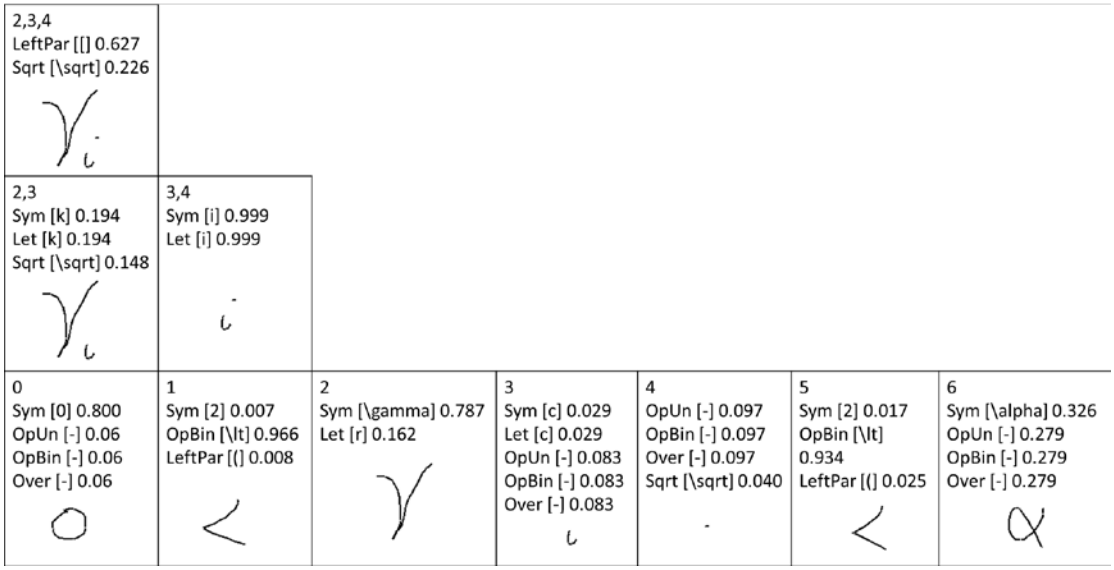
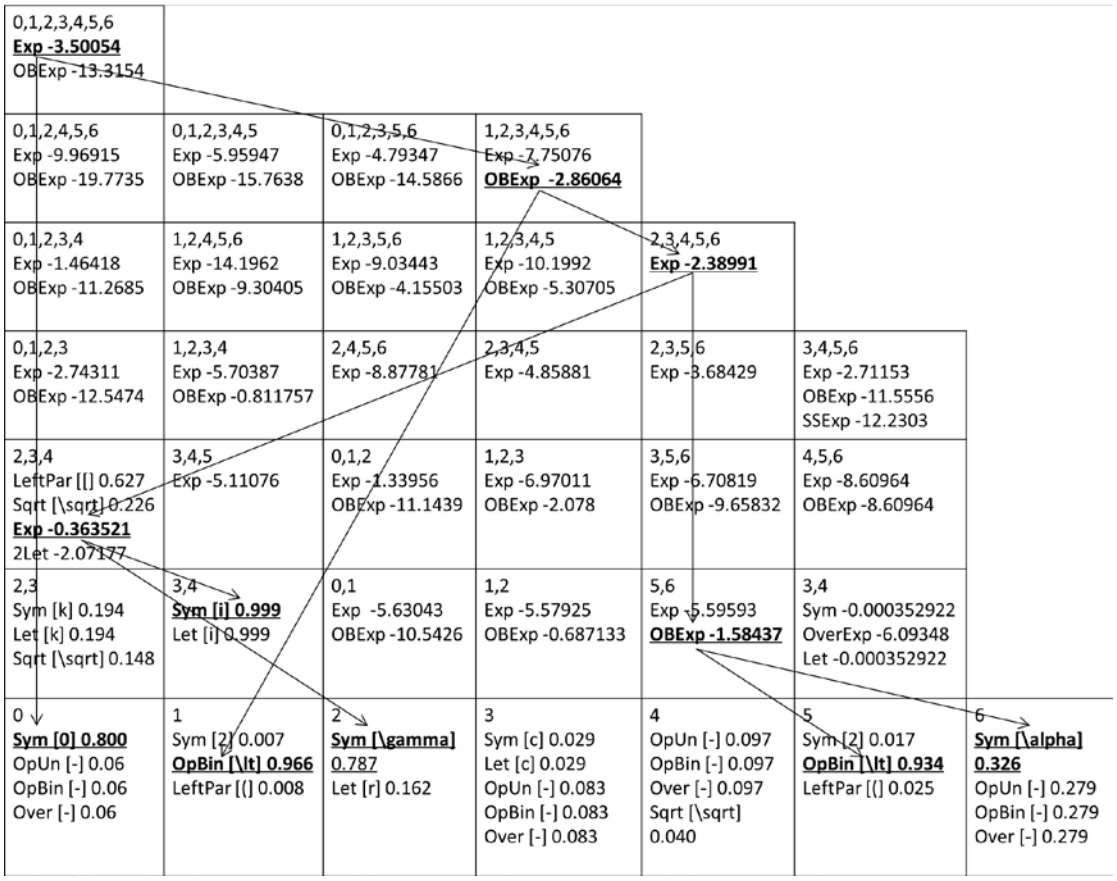


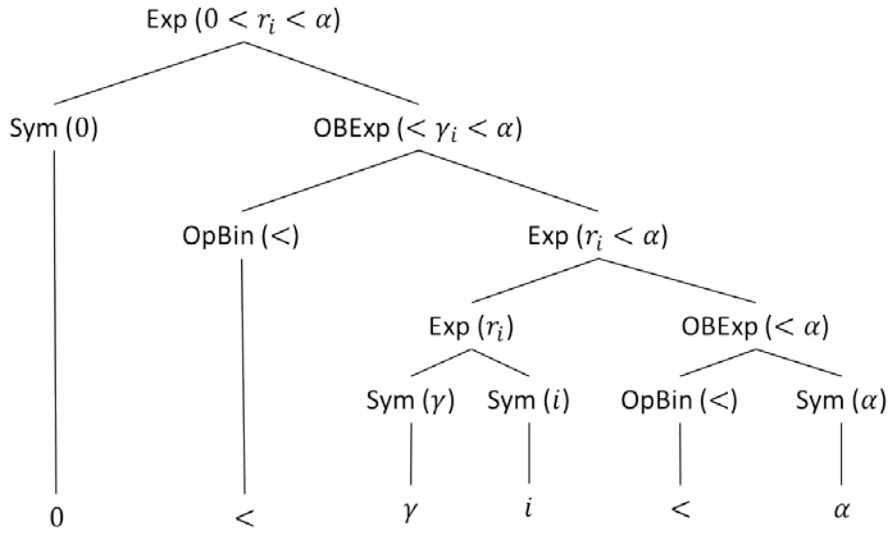
Figure 3.28 Initialization of table

0,1,2,3,4,5,6 Exp -3.50054 OBExp -13.3154						
0,1,2,4,5,6 Exp -9.96915 OBExp -19.7735	0,1,2,3,4,5 Exp -5.95947 OBExp -15.7638	0,1,2,3,5,6 Exp -4.79347 OBExp -14.5866	1,2,3,4,5,6 Exp -7.75076 OBExp -2.86064			
0,1,2,3,4 Exp -1.46418 OBExp -11.2685	1,2,4,5,6 Exp -14.1962 OBExp -9.30405	1,2,3,5,6 Exp -9.03443 OBExp -4.15503	1,2,3,4,5 Exp -10.1992 OBExp -5.30705	2,3,4,5,6 Exp -2.38991		
0,1,2,3 Exp -2.74311 OBExp -12.5474	1,2,3,4 Exp -5.70387 OBExp -0.811757	2,4,5,6 Exp -8.87781	2,3,4,5 Exp -4.85881	2,3,5,6 Exp -3.68429	3,4,5,6 Exp -2.71153 OBExp -11.5556 SSExp -12.2303	
2,3,4 LeftPar [[] 0.627 Sqrt [\sqrt] 0.226 Exp -0.363521 2Let -2.07177	3,4,5 Exp -5.11076	0,1,2 Exp -1.33956 OBExp -11.1439	1,2,3 Exp -6.97011 OBExp -2.078	3,5,6 Exp -6.70819 OBExp -9.65832	4,5,6 Exp -8.60964 OBExp -8.60964	
2,3 Sym [k] 0.194 Let [k] 0.194 Sqrt [\sqrt] 0.148	3,4 Sym [i] 0.999 Let [i] 0.999	0,1 Exp -5.63043 OBExp -10.5426	1,2 Exp -5.57925 OBExp -0.687133	5,6 Exp -5.59593 OBExp -1.58437	3,4 Sym -0.000352922 OverExp -6.09348 Let -0.000352922	
0 Sym [0] 0.800 OpUn [-] 0.06 OpBin [-] 0.06 Over [-] 0.06	1 Sym [2] 0.007 OpBin [\lt] 0.966 LeftPar [(] 0.008	2 Sym [\gamma] 0.787 Let [r] 0.162	3 Sym [c] 0.029 Let [c] 0.029 OpUn [-] 0.083 OpBin [-] 0.083 Over [-] 0.083	4 OpUn [-] 0.097 OpBin [-] 0.097 Over [-] 0.097 Sqrt [\sqrt] 0.040	5 Sym [2] 0.017 OpBin [\lt] 0.934 LeftPar [(] 0.025	6 Sym [\alpha] 0.326 OpUn [-] 0.279 OpBin [-] 0.279 Over [-] 0.279

Figure 3.29 The complete parsing table



(a)



(b)

Figure 3.30 (a) Obtain a parsing tree by going through the table from top to bottom. (b) The obtained parsing tree.

Chapter 4. Databases and Experiments

We have developed a system for online handwritten mathematical expression recognition. Now we describe the experiments that we carried out to test our developed system. Firstly, we describe the databases that we used. Then some experiments and result are presented.

4.1 Database

We used two database, CROHME 2011 (Part-Ⅱ) and CROHME 2012 (Part-Ⅲ) (see Table 4.1). There are few restrictions on the grammars of these two databases. For example, there is no limit on recursions of operations like sum, product, function call, fraction, root, sub/superscript on symbols, etc. However, CROHME 2012 (Part-Ⅲ) covers more terminal symbols than CROHME 2011 (Part-Ⅱ), and the grammar in CROHME 2012 (Part-Ⅲ) is more complicated.

Both of them are from CROHME (Competition on Recognition of Online Handwritten Mathematical Expressions). This competition was organized by ICDAR (International Conference on Document Analysis and Recognition) on 2011 and 2012. An overview of this competition is in [23, 24].

Table 4.1 Description for database CROHME 2011 and CROHME 2012

Database	Dataset Type	Number of expressions	Number of Terminals
CROHME 2011 Part-Ⅱ	Training	921	57
	Test	348	
CROHME 2012 Part-Ⅲ	Training	1336	75
	Test	488	

In these two databases, the ink corresponding to each expression is stored in an InkML file. An InkML file mainly contains three kinds of information: (1) the ink: a set of traces made of points; (2) the symbol level ground truth: the segmentation and label information of each symbol of the expression; and (3) the expression level

ground truth: the MathML structure of the expression.

The two levels of ground truth information (at the symbol as well as at the expression level) are entered manually. Furthermore, some general information is added in the file: (1) the channels (here, X and Y); (2) the writer information (identification, handedness (left/right), age, gender, etc.), if available; (3) the LaTeX ground truth (without any reference to the ink and hence, easy to render); (4) the unique identification code of the ink (UI), etc.

The InkML format enables to make references between the digital ink of the expression, its segmentation into symbols and its MathML representation. An example of an InkML file for the expression $A \times B$ is shown as Figure 4.1. It contains 6 strokes for 3 symbols (two for the each symbol). Note that the traceGroup with identifier xml:id="7" has references to the 2 corresponding strokes of symbol "A", as well as to the MathML part with identifier xml:id="A_1". Thus, the stroke segmentation of a symbol can be linked to its MathML representation.

```
<ink xmlns="http://www.w3.org/2003/InkML">
<traceFormat>
<channel name="X" type="decimal"/>
<channel name="Y" type="decimal"/>
</traceFormat>
<annotation type="truth">$A\times B$</annotation>
<annotation type="UI">2012_IVC_CROHME_F01_E0012</annotation>
<annotation type="copyright">LUNAM/IRCCyN</annotation>
<annotation type="writer">CROHME01</annotation>
<annotationXML type="truth" encoding="Content-MathML">
    <math xmlns='http://www.w3.org/1998/Math/MathML'>
        <mrow>
            <mi xml:id="A_1">A</mi>
            <mrow>
                <mo xml:id="\times_1">\times</mo>
                <mi xml:id="B_1">B</mi>
            </mrow>
        </mrow>
    </math>
</annotationXML>
<trace id="0">
1.10641 6.55641, … , 1.15034 6.55598
</trace>
…
<trace id="5">
1.23863 6.51972, … , 1.2446 6.54873
</trace>
<traceGroup xml:id="6">
    <annotation type="truth">Segmentation</annotation>
    <traceGroup xml:id="7">
        <annotation type="truth">A</annotation>
```

```
        <traceView traceDataRef="0"/>
        <traceView traceDataRef="1"/>
        <annotationXML href="A_1"/>
    </traceGroup>
    …
</traceGroup>
</ink>
```

Figure 4.1 Example of InkML format

4.2 Experiments

According to CROHME 2011 and 2012, four aspects should be measured to evaluate the performance of a system on online handwritten mathematical expression recognition [23, 24]. They are (1) ST_Rec: the stroke classification rate, representing the percentage of strokes with the correct symbol, (2) SYM_Seg: the symbol segmentation rate, defining the percentage of symbols correctly segmented, (3) SYM_Rec: the symbol recognition rate, computing the performance of the symbol classifier when considering only the correct segmented symbols. The last measurement is (4) EXP_Rec: the expression recognition rate, which informs the percentage of expressions totally correctly recognized. This is a very challenging indicator since the slightest error anywhere in the expression prevents to count it. In order to have a better insight of the capacity of the respective systems, [24] also extended this indicator with (5) EXP_Rec_1, _2, _3, giving the percentage of expressions recognized with at most 1 error, 2 errors and 3 errors (in terminal symbols or in MathML node tags) given that the tree structure is correct.

We tested our system on the test dataset of both databases. The results are reported in Table 4.2.

Firstly, we compared the values in horizontal direction. The first three value ST_Rec, SYM_Seg, SYM_Rec stay high, showing that the segmentation hypothesis generator and symbol classifier work quite well. However, the value EXP_Rec is very low on both datasets. We tried to explore the reason. On one hand, EXP_Rec is a very strict indicator as is explained above. Even only one single error in the expression will prevent to count it. On the other hand, the system cannot correctly analyze the structure even though most of segmentation and symbols are recognized. It implies that the model of spatial relation needs to be improved.

We also noted that a big gap exists between EXP_Rec and EXP_REC_1, showing that many expressions go wrong because of only one error. As long as correcting these single errors, our system will have much improvement. On the other hand, the narrow differences between EXP_REC_2 and EXP_REC_3 show that when more errors go wrong, it is difficult to improve the accuracy. Therefore, to improve our system, we can put more focus on the one-error expressions.

Then, we compared the values in vertical direction. All the values of CROHME 2012 (Part-Ⅲ) are lower than that of CROHME 2011 (Part-Ⅱ), proving that the

grammar in CROHME 2012 (Part-III) is more complicated. It shows that our system cannot handle complicated expressions very well.

Table 4.2 Main results on the test dataset of CROHME 2011 and CROHME 2012

Dataset	*ST_Rec*	*SYM_Seg*	*SYM_Rec*	*EXP_Rec*	*EXP_Rec_1*	*EXP_Rec_2*	*EXP_Rec_3*
CROHME 2011 Part-II	82.03%	87.94%	93.23%	30.46%	44.83%	48.85%	50.00%
CROHME 2012 Part-III	75.31%	82.56%	92.53%	17.62%	28.28%	31.56%	32.79%

To compare our system with the participants of CROHME 2011 and CROHME 2012, their results are shown in Table 4.3 [24]. As we can see, our system outperforms all the participants of CROHME 2011 at every aspect. However, we ranked only five out of eight in CROHME 2012.

Table 4.3 Comparison between our system and the participating systems of CROHME 2011 and CROHME 2012

CROHME 2011 Part-II				
System	***ST_Rec***	***SYM_Seg***	***SYM_Rec***	***EXP_Rec***
I	51.58%	56.50%	91.29%	2.59%
II	22.11%	28.25%	83.76%	0.29%
III	78.38%	87.82%	92.56%	19.83%
IV	52.28%	78.77%	78.67%	0.00%
V	70.79%	84.23%	87.16%	22.41%
Our system	**82.03%**	**87.94%**	**93.23%**	**30.46%**
CROHME 2012 Part-III				
System	***ST_Rec***	***SYM_Seg***	***SYM_Rec***	***EXP_Rec***
I	79.85%	91.95%	86.25%	22.75%
II	55.75%	71.21%	84.97%	3.69%
III	78.94%	87.75%	91.38%	25.61%
IV	72.12%	87.51%	87.62%	9.43%
V	45.42%	59.20%	84.27%	4.92%
VI	86.41%	95.56%	91.17%	40.16%
VII (Winner)	**95.75%**	**98.84%**	**96.85%**	**62.50%**
Our system	75.31%	82.56%	92.53%	17.62%

Another interesting analysis concerns the distribution of errors with respect to the size of the expressions. Of course, the longer the expressions, the harder it is to recognize them. Figure 4.3 illustrates this behavior. Our system only achieved 44.6% and 26.8% recognition rates (on CROHME 2011 and CROHME 2012, respectively) among the shortest expressions. This kind of short expressions is also our target of improvement.

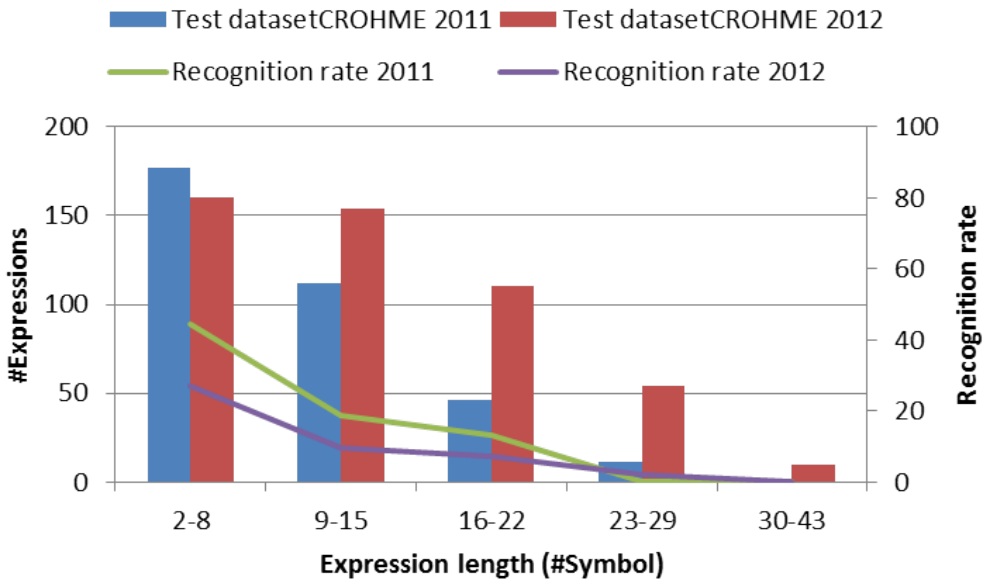


Figure 4.2 Recognition rates (EXP_Rec) with respect to the expression length

As is discussed above, generating hypotheses covering all the correct segmentation is the first task in the recognition process. If we don't have correct candidates at first, it is impossible to achieve a correct solution at the end. We did another experiment to evaluate the performance of the segmentation hypothesis generator. In this experiment, recall is measured. Two terms are related to the value $recall$: $true\ positives$ and $false\ negatives$. The term $true\ positives$ refers to hypotheses that are actually correct segmentation. The term $false\ negatives$ refers to the correct segmentations that the generator missed to create. Recall is then defined as:

$$recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

$Recall$ is the capability of successfully retrieving the correct segmentations. Ideally, $recall$ equals to be 1, meaning that the hypotheses covers all the correct segmentations. The experimental results are shown in Table 4.4. Recalls in two stages were tested. Before classifying, recall is almost 1, showing that the generated hypotheses cover almost all the correct segmentations. However, recall went down to 0.97 after classifying on both databases. That is to say some correct segmentations were removed by the classifier. It is probably because of the classifier misrecognizing those correct segmentations to be 'junk'. Thus, retraining the classifier to recognize

the junk class is another potential to improve our system.

Table 4.4 Recall for segmentation hypotheses

Dataset	Recall	
	Before classifying	After classifying
CROHME 2011	0.9954	0.9766
CROHME 2012	0.9978	0.9772

Chapter 5. Conclusions

In this report, we presented an on-line handwritten mathematical expression recognition system based on 2D-SCFG. We defined a grammar covering a wide range of expressions. We proposed a segmentation hypothesis generator using a searching area to combine closer strokes. CYK parsing algorithm was used to analyze the structure. For different kinds of spatial relation, we used some simple geometric features to model them. Finally, we performed several experiments. Our system had a moderate performance.

For future work, there are a lot of issues to study.

5.1 Grammar Learning

In this work, the grammar is defined manually and every production rule has a fixed probability. A very interesting objective is to learn the probabilities of the production rules of the SCFG from a training dataset.

5.2 Searching Area

We used searching area for both generating multiple-stroke hypothesis and building subexpressions during CYK parsing. The size of searching area has an important impact. Smaller size discards some possible cases and finally cannot guarantee the optimal solution. Bigger size introduces too many cases and finally increases the computational time. Thus, the size of searching area is hard to determine.

For expressions with a short symbol interval, it takes a long time to recognize because the searching area discovers too many cases. For expressions with a long symbol interval, we could not fully recognize since searching area is not big enough to build new subexpression. As a result, the gap between symbols should be taken into account when we determine the size of searching area.

5.3 Spatial Relation

In Section 3.5, we explained the probability distributions used to model the spatial relation. But these models are very simple. It is necessary to employ more

features. For example, the baseline information should contribute in the performance of the system.

The spatial distributions are defined manually. It is also very interesting to be able to automatically learn the parameters of these distributions.

5.4 Complexity

Currently, it takes nearly 30 hours to perform our system on a test dataset of 488 expressions. When the number of symbols of the expression is large, the cost of time is very expensive. It is still far from practical. For that reason, it is necessary to reduce this computational cost.

Using thresholds to remove improbable hypotheses helps to improve reducing the computational time, but it cannot guarantee that the optimal solution is achieved.

The complexity of the parsing is relative to the number of strokes. The parsing table becomes bigger as the expression has more strokes. As a result, another possible solution is to adjust the input of CYK parsing algorithm from single stroke connected strokes. It probably reduces the size of parsing table because the number of connected strokes is far less than the number of strokes.

Reference

[1] Michael Shilman, Hanna Pasula, Stuart Russell, Richard Newton, Statistical Visual Language Models for Ink Parsing, American Association for Artificial Intelligence, 2000.

[2] F. Perraud, C. Viard-Gaudin, E. Morin, P.M. Lallican, “Statistical Language Models for On-Line Handwriting Recognition”, IEICE Transactions on Information and Systems/Document Image Understanding and Digital Document, Vol.E88-D No.8 pp.1807-1814, 2005.

[3] N. Chomsky, “Three models for the description of language,” IRE Transactions on Information Theory, vol. 2, pp. 113–124, 1956.

[4] D. Blostein and A. Grbavec, “Recognition of mathematical notation,” *Handbook of character recognition and document image analysis*, pp. 557–582, 1997.

[5] K.-F. Chan and D.-Y. Yeung, “Mathematical expression recognition: a survey,” *International Journal on Document Analysis and Recognition*, vol. 3, no. 1, pp. 3–15, Aug. 2000.

[6] A. Awal, H. Mouchère, and C. Viard-Gaudin, “A global learning approach for an online handwritten mathematical expression recognition system,” *Pattern Recognition Letters*, Nov. 2012.

[7] R. Yamamoto and S. Sako, “On-line recognition of handwritten mathematical expressions based on stroke-based stochastic context-free grammar,” *tenth International Workshop on Frontiers in Handwriting Recognition*, 2006.

[8] Francisco Álvaro, Joan-Andreu Sánchez, “Recognition of On-line Handwritten Mathematical Expressions Using 2D Stochastic Context-Free Grammars and Hidden Markov Models,” *Pattern Recognition Letters*, 2012.

[9] D. Průša and V. Hlaváč, “2D context-free grammars: Mathematical formulae recognition,” *Proceedings of the Prague Stringology Conference*, vol. 6, pp. 77–89, 2006.

[10] D. Průša and V. Hlaváč, “Structural construction for on-line mathematical formulae recognition,” *Progress in Pattern Recognition, Image Analysis and Applications*, pp. 317–324, 2008.

[11] Jan Stria, Daniel Průša, Václav Hlaváč, “Combining Structural and Statistical Approach to Online Handwritten Math Recognition.”

[12] Álvaro, F.; Sanchez, J.-A.; Benedi, J.-M., "Recognition of Printed Mathematical Expressions Using Two-Dimensional Stochastic Context-Free Grammars," *Document Analysis and Recognition (ICDAR), 2011 International Conference on* , vol., no., pp.1225,1229, 18-21 Sept. 2011

[13] S. MacLean, G. Labahn, and E. Lank, “Grammar-based techniques for creating ground-truthed sketch corpora,” *International Journal on Document Analysis and Recognition (IJDAR)*, vol. 14, pp. 1–21, 2011.

[14] M. Shilman, P. Viola, and K. Chellapilla, “Recognition and Grouping of Handwritten Text in Diagrams and Equations,” *Ninth International Workshop on Frontiers in Handwriting Recognition*, pp. 569–574, 2004.

[15] A. Kosmala and G. Rigoll, “On-line handwritten formula recognition using hidden Markov models and context dependent graph grammars,” *Document Analysis and Recognition, 1999. ICDAR’99. Proceedings of the Fifth International Conference*, pp. 107–110, 1999.
[16] M. Shilman, and P. Viola, “Learning nongenerative grammatical models for document analysis,” *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference*, vol. 2, pp. 1–7, 2005.
[17] E. Miller and P. Viola, “Ambiguity and constraint in mathematical expression recognition,” *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, vol. 784–791, 1998.
[18] Liang, P.; Narasimhan, M.; Shilman, M.; Viola, P., "Efficient geometric algorithms for parsing in two dimensions," *Document Analysis and Recognition, 2005. Proceedings. Eighth International Conference on* , vol., no., pp.1172,1177 Vol. 2, 29 Aug.-1 Sept. 2005
[19] a. Grbavec and D. Blostein, “Mathematics recognition using graph rewriting,” *Proceedings of 3rd International Conference on Document Analysis and Recognition*, vol. 1, pp. 417–421, 1995.
[20] P. Garcia and B. Coüasnon, “Using a generic document recognition method for mathematical formulae recognition,” *Graphics Recognition Algorithms and Applications*, pp. 236–244, 2002.
[21] S. MacLean and G. Labahn, “Recognizing handwritten mathematics via fuzzy parsing,” Tech. Rep. CS-2010-13, School of Computer Science, University of Waterloo, 2010. 3, 2010.
[22] J. Fitzgerald, “Structural analysis of handwritten mathematical expressions through fuzzy parsing,” ACST’06: Proceedings of the Second IASTED International Conference on Advances in Computer Science and Technology, pp. 151–156, 2006.
[23] H. Mouchère and C. Viard-Gaudin, “Crohme2011: Competition on recognition of online handwritten mathematical expressions,” International Conference on Document Analysis and Recognition (ICDAR), 2011, pp. 1497–1500, 2011.
[24] H. Mouchère and C. Viard-Gaudin, “ICFHR 2012 Competition on Recognition of On-Line Mathematical Expressions (CROHME 2012),” International Conference on Frontiers in Handwriting Recognition (ICFHR), 2012, pp. 811–816, 2012.